THE UNIVERSITY
of EDINBURGH

Progressive Load Balancing of Asynchronous Algorithms

Justs Zarins

Doctor of Philosophy

Institute of Computing Systems Architecture

School of Informatics

University of Edinburgh

2021

Abstract

Massively parallel supercomputers are susceptible to variable performance due to factors such as differences in chip manufacturing, heat management and network congestion. As a result, the same code with the same input can have a different execution time from run to run. Synchronisation under these circumstances is a key challenge that prevents applications from scaling to large problems and machines.

Asynchronous algorithms offer a partial solution. In these algorithms fast processes are not forced to synchronise with slower ones. Instead, they continue computing updates, and moving towards the solution, using the latest data available to them, which may have become stale (i.e. the data is a number of iterations out of date compared to the most recent version). While this allows for high computational efficiency, the convergence rate of asynchronous algorithms tends to be lower than synchronous algorithms due to the use of stale values. A large degree of performance variability can eliminate the performance advantage of asynchronous algorithms or even cause the results to diverge.

To address this problem, we use the unique properties of asynchronous algorithms to develop a load balancing strategy for iterative convergent asynchronous algorithms in both shared and distributed memory. The proposed approach – Progressive Load Balancing (PLB) – aims to balance progress levels over time, rather than attempting to equalise iteration rates across parallel workers. This approach attenuates noise without sacrificing performance, resulting in a significant reduction in progress imbalance and improving time to solution.

The developed method is evaluated in a variety of scenarios using the asynchronous Jacobi algorithm. In shared memory, we show that it can essentially eliminate the negative effects of a single core in a node slowed down by 19%. Work stealing, an alternative load balancing approach, is shown to be ineffective. In distributed memory, the method reduces the impact of up to 8 slow nodes out of 15, each slowed down by 40%, resulting in 1.03×–1.10× reduction in time to solution and 1.11×–2.89× reduction in runtime variability. Furthermore, we successfully apply the method in a scenario with real faulty components running 75% slower than normal. Broader applicability of progressive load balancing is established by emulating its application to asynchronous stochastic gradient descent where it is found to improve both training time and the learned model's accuracy.

Overall, this thesis demonstrates that enhancing asynchronous algorithms with PLB is an effective method for tackling performance variability in supercomputers.

Lay Summary

Supercomputers are made up of many normal computers connected together in a closely integrated network. They are used in many areas ranging from simulation of individual molecules for medical uses to modelling weather events to create the next day's forecast. A large problem is broken down into small pieces or steps that can be worked on simultaneously. Then, importantly, all parts of the supercomputer work together, as a team, to solve the problem at hand. In the process, information and partial solutions need to be exchanged between the parts in order to coordinate and to put together the whole solution to the problem.

During the operation of a supercomputer all of its parts do not run at exactly the same speed. This may be caused by small differences in how the parts were manufactured or something that happens during operation, for example too much communication taking place at one time, thus causing congestion. Since the parts are cooperating closely (they are "synchronous"), any delay in one part causes all others to wait and waste time. However, in some applications it is mathematically possible to progress towards the solution with partially up-to-date information, so the requirement to wait for slower parts can be dropped; such applications are called "asynchronous". While this is faster, one has to be careful not to use information that is excessively out-of-date (i.e. stale).

In this thesis we develop a method which prevents asynchronous applications from working with excessively stale information while retaining the performance benefits of asynchronous computation. Our method – "progressive load balancing" – moves work between areas of the supercomputer in a way which speeds up progress on parts of the work that have fallen behind and slows down progress on parts that have run ahead. As a result, when information is exchanged between components, it is less stale on average than it would be without load balancing. We test our method in a range of settings on applications that are used to simulate physical processes and to train artificial intelligence. We found that our method mitigates variable speed between the parts of a supercomputer to a great extent, which leads to quicker arrival at the solution and less wasted computer effort. As a result, supercomputers can be used more efficiently and more work can be done in less time.

Acknowledgements

First of all, I would like to thank my supervisor Michèle Weiland for her guidance and help throughout my PhD studies. I would also like to thank my other supervisors Lorna Smith and Bjoern Franke for their input on the project. Next, I would like to thank my PPar colleagues for the academic discussions and companionship on the long road. Finally, I would like to thank the programme's leaders and EPCC for providing support and many opportunities for development.

This work was supported by the Engineering and Physical Sciences Research Council (grant EP/L01503X/1), EPSRC Centre for Doctoral Training in Pervasive Parallelism at the University of Edinburgh, School of Informatics.

This work used the ARCHER UK National Supercomputing Service (`http://www.archer.ac.uk`). This work also used the Cirrus UK National Tier-2 HPC Service at EPCC (`http://www.cirrus.ac.uk`) funded by the University of Edinburgh and EPSRC (EP/P020267/1).

The Fulhame HPE Apollo 70 system is supplied to EPCC as part of the Catalyst UK programme, a collaboration with Hewlett Packard Entreprise, Arm and SUSE to accelerate the adoption of Arm based supercomputer applications in the UK.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified. Some of the material used in this thesis has been published in the following papers:

[1] Justs Zarins and Michèle Weiland.
Progressive load balancing of asynchronous algorithms.
Proceedings of the Seventh Workshop on Irregular Applications: Architectures and Algorithms (IA3), 2017.

[2] Justs Zarins and Michèle Weiland.
Progressive load balancing in distributed memory.
Proceedings of the International Conference on Parallel Computing (PARCO), 2019.

(*Justs Zarins*)

Table of Contents

1 Introduction **1**

1.1 Contributions 3

1.2 Thesis structure 4

2 Background and literature review **7**

2.1 High performance computing 7

2.1.1 Scaling 8

2.1.2 Synchronous algorithms 9

2.2 Performance variability 9

2.2.1 Types and sources of performance variability 9

2.2.2 Impact of performance variability 11

2.2.3 Ways to reduce or mitigate performance variability 12

2.3 Asynchronous algorithms – an alternative 14

2.3.1 Basics of asynchronous algorithms 15

2.3.2 Detailed example of asynchrony 17

2.3.3 The application of asynchronous algorithms 20

2.3.4 Progress variability in asynchronous algorithms and ways to address it 21

2.4 Summary 26

3 Designing a load balancing scheme **29**

3.1 Simulator 29

3.1.1 Simulator design and implementation 30

3.1.2 Designing the load balancing algorithm 33

3.2 Load balancing details beyond simulation 43

3.2.1 Implementation 43

3.2.2 PLB parameters 45

3.2.3 Variants due to hardware 47
3.3 Summary 47

4 PLB in shared memory **49**
4.1 Methodology and experiments 49
4.1.1 PLB and Jacobi implementation 49
4.1.2 Evaluation metrics 50
4.1.3 Measurement and hardware setup 51
4.1.4 Test problem and load balancing settings 51
4.1.5 Simulating noise 53
4.2 Evaluation 54
4.2.1 Spread reduction 55
4.2.2 Iteration rate 58
4.2.3 Time to solution improvement 58
4.2.4 Comparison with work stealing 61
4.3 Summary 64

5 PLB in distributed memory **65**
5.1 Extending PLB to distributed memory 65
5.1.1 IPLB 65
5.1.2 DPLB 66
5.1.3 Implementation 66
5.2 Experiments 68
5.3 Evaluations 70
5.3.1 IPLB 71
5.3.2 DPLB 77
5.3.3 Profiling 83
5.4 Summary 85

6 Extended use cases of DPLB **87**
6.1 DPLB in the presence of real performance variation 87
6.1.1 Results 89
6.2 DPLB applied to Stochastic Gradient Descent 94
6.2.1 Setup of experiments 96
6.2.2 Experiment results 98
6.3 Practical considerations of applying DPLB to other algorithms 102

6.4 Summary 102

7 Conclusion **105**

7.1 Discussion, limitations and future work 106

7.2 Summary 109

A Distributed asynchronous Jacobi with DPLB pseudocode **111**

Bibliography **113**

Glossary

ASGD Asynchronous Stochastic Gradient Descent

ASync Asynchronous

CPU Central Processing Unit

DPLB Distributed Progressive Load Balancing

GPU Graphics Processing Unit

HPC High Performance Computing

IPLB Independent Progressive Load Balancing

MPI Message Passing Interface

NIC Network Interface Controller

OS Operating System

PE Processing Element

PLB Progressive Load Balancing

RMA Remote Memory Access

SGD Stochastic Gradient Descent

SSync Semi-Synchronous

Sync Synchronous

TTS Time To Solution

WI Work Item

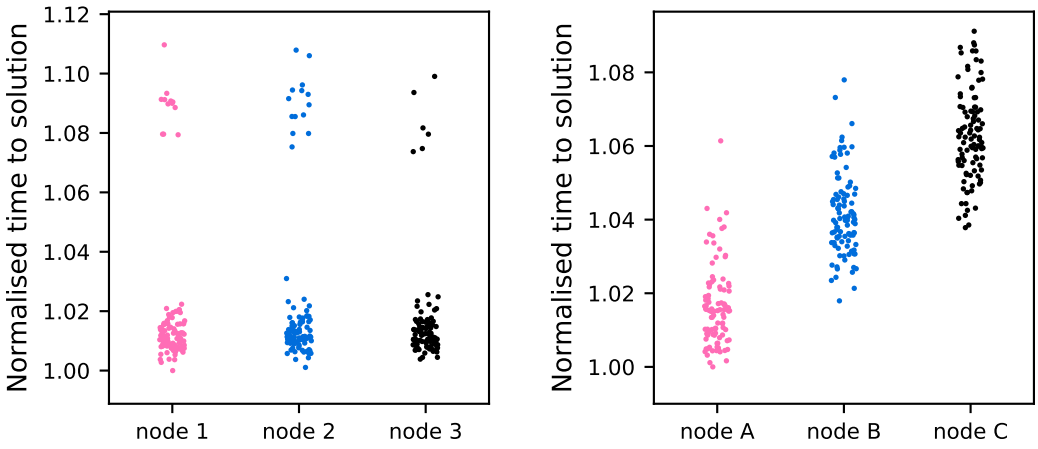
Chapter 1

Introduction

As supercomputers are growing in size, scaling tightly-coupled applications efficiently is becoming more difficult. Many applications running on supercomputers progress in a bulk synchronous manner [3] with multiple computation phases separated by synchronisation points, such as barriers. This hinders scalability, because bulk synchronisation increases an application's sensitivity to "performance variability" leading to an increase in the time spent in the synchronisation points.

Performance variability or "noise" is the phenomenon where repeated executions of an application with the same inputs and on the same hardware complete with significantly variable runtime. Performance variability can also occur between different parts of a single application running in a distributed manner; in this scenario workload imbalance exacerbates the problem. The root cause of performance variability ranges from operating system (OS) jitter to chip manufacturing differences and network congestion (see Section 2.2.1). Noise affects even high-end high performance computing (HPC) machines like ARCHER [4] and Cirrus [5] as shown in Figure 1.1. These plots show the results of repeatedly running the synchronous Jacobi algorithm (see Section 2.3.2) in shared memory until convergence. Performance variability of around 10% can be seen on these production machines and it manifests itself in two different patterns. In Figure 1.1a all nodes perform similarly, but there are outliers in each, likely due to random noise accumulation. In Figure 1.1b the performance spread is smaller on each node, but there is a constant shift factor between nodes. This pattern could be caused by component binning [6] resulting in small but constant speed differences between nodes.

Traditional bulk synchronous algorithms face difficulties coping with variable machine performance because the global progress rate is limited by the slowest compo-



(a) ARCHER **(b)** Cirrus

Figure 1.1: Performance variability within and across different nodes on two HPC machines. Each point is a run of the same application (synchronous Jacobi algorithm in shared memory) with the same settings. Times are normalised to the fastest run on each machine.

nent. Additionally, as the scale of computation increases and more nodes participate, the chance of encountering noise also increases, as does the resulting wasted time. To continue the growth of parallel computing one has to tackle performance variability.

A potential solution exists in the form of "asynchronous" or "chaotic" algorithms [7]. Commonly these are iteratively convergent algorithms with the key feature that threads are allowed to compute using whatever latest data is available to them, which might be "stale", instead of waiting for other threads to catch up (see Section 2.3.1 for more details; also note that here we are not considering task based parallelism). Existing applications of this methodology show good performance and fault tolerance with respect to their synchronous counterparts [8, 9]. Asynchronous algorithms exhibit superior hardware efficiency because wait time due to random noise does not accumulate at synchronisation points. Additionally, communications can be spread out over time whereas barriers lead to "bunched up" communications that become a performance bottleneck. Network injection points have finite capacity and a sudden burst of communication can hit this limit, resulting in congestion, which in turn harms performance.

However, asynchronous algorithms are not totally immune from noise. While asynchrony removes the computational cost of requiring all data to arrive at the same time,

a different cost takes its place – progress imbalance. This is natural because synchronisation points exist to coordinate progress. An imbalance in progress can result in slower convergence [10] or even failure to converge [11], as old data is used for updates. This can be countered by putting a strict bound on how stale data is allowed to be, but at a cost to performance. Eventually, the performance can end up again being limited by the component that has made the least progress.

This thesis demonstrates that a novel load balancing approach can reduce the negative effects of performance variability on asynchronous algorithms but without losing their superior scaling properties. Existing solutions to progress variation are either not readily applicable to asynchronous algorithms, or they are specific to an algorithm or architecture (see Section 2.3.4). This analysis demonstrates the more general application of load balancing to asynchronous algorithms in order to improve their suitability for supercomputing applications.

Specifically, we use what makes asynchronous algorithms unique – allowing staleness – to formulate a load balancing technique specific to this context called progressive load balancing (PLB). PLB balances asynchronous algorithms *over time* as opposed to balancing instantaneously. Subdomains of the problem are periodically moved between computing threads to slow down the update rates of subdomains that are ahead and to speed up the update rates of subdomains that are falling behind. This allows greater flexibility in managing noise than the fine-tuning of iteration rates and thus can be adjusted to the needs of each application. As a result we limit progress imbalance without adding a large overhead. We apply this technique both in the shared and distributed memory setting.

1.1 Contributions

This thesis makes the following contributions:

- We design and implement a simulator of iterative asynchronous algorithms. We then use the simulator to incrementally develop a load balancing method that addresses both workload and hardware variability (Chapter 3).

- We design and implement a load balancing algorithm for iterative asynchronous algorithms in shared memory – progressive load balancing (PLB). We show that update spread is bounded under a variety of scenarios using PLB and that the

overhead of balancing is small (1%–7%) in most cases. We then show that asynchrony with PLB minimises the impact of a 19% slowed down core on a compute node. As a consequence, asynchrony with PLB gives a 5%–25% speedup in time to solution over other synchronisation methods. Finally, we compare PLB to work stealing (Chapters 3 and 4) and find the latter ineffective for balancing asynchronous algorithms. The evaluation of PLB has been published in [1].

- We design and implement two extensions of PLB to distributed memory – independent PLB (IPLB) and distributed PLB (DPLB). We show that IPLB is useful in the situation where noise is symmetric across nodes, but DPLB can handle any noise pattern. We demonstrate successful mitigation of the impact of up to 8 nodes out of 15 slowed down by 40%: a reduction of 1.08×–4.05× in global progress variability and 1.03×–1.10× reduced time to solution as well as time to solution variability reduction of 1.11×–2.89×. We also profile the overhead of distributed load balancing (finding negligible runtime overhead) and apply DPLB to a scenario with real performance variability (Chapters 5 and 6). The evaluation of DPLB has been published in [2].

- We evaluate the application of DPLB to asynchronous stochastic gradient descent by emulating DPLB. We show that DPLB can mitigate 1 out of 4 nodes slowed down by 40%, thus reducing runtime by 67% (in comparison to no load balancing) and retaining or improving model accuracy. We also discuss some practical considerations on how to apply and evaluate DPLB to other applications (Chapter 6).

1.2 Thesis structure

In **Chapter 2** we describe the technical background of performance variability and asynchronous algorithms. We then evaluate existing approaches of addressing noise and arrive at dynamic load balancing as our method to tackle the problem. **Chapter 3** describes an asynchronous algorithm simulator and how we used it to evaluate load balancing methods which lead up to progressive load balancing. In **Chapter 4** we evaluate PLB in shared memory and compare it against work stealing. In **Chapter 5** we extend PLB to distributed memory (IPLB and DPLB) and present a detailed evaluations. In **Chapter 6** we show the application of DPLB to a real performance variability scenario. We also evaluate the application of emulated DPLB to asynchronous

stochastic gradient descent. **Chapter 7** presents a summary of this thesis and a detailed discussion of its limitations as well as future work directions.

Chapter 2

Background and literature review

In this Chapter we expand on the problem addressed in this thesis – synchronisation in the presence of performance variability – and provide the technical background of the associated concepts. We also examine the existing solutions and work related to our study, identifying their limitations, and set the scene for a new solution.

2.1 High performance computing

Ever since the creation of the first computers, there have been efforts to make them run faster and faster in order to do more work in less time. For a long time it was possible to simply buy the latest processors and expect them to run at a higher clock speed thus delivering higher performance with little effort required from the software side. This was made possible by the transistor count in the same chip area doubling approximately every two years (Moore's law [12]) due to technological advances allowing to shrink individual transistors. The smaller components were able to operate quicker, thus increasing the clock speed of processors. Simultaneously, the voltage requirements for individual transistors decreased due to their shrinking size, leading to constant power per chip area (Dennard scaling [13]). This meant chips became faster within the same power envelope and it remained possible to cool them efficiently. Eventually, around 2006, physical limitations like leakage currents [14] put an end to Dennard scaling and clock frequency of cores stopped increasing. At this point the method of choice for getting more performance from a single chip was to put multiple computing cores on it, running in parallel.

Increased performance is also harnessed by networking multiple machines together to form a cluster. These machines can work together, in parallel, to solve a single prob-

lem. Even individual instructions can operate on vectors of values, performing multiple calculations at the same time. Thus, modern supercomputers operate in parallel at every level, from nodes to instructions, all in the pursuit of ever greater speed.

Scaling up computational capability unlocks new scientific and industrial research possibilities by increasing the fidelity and scope of simulation. This benefits many important application areas such as scientific simulation, industrial engineering as well as operational uses, e.g. daily global weather prediction [15]. These applications have immense economic and societal value, so there is always an appetite for more computational capability. This has lead to the growth of parallelism resulting in the most powerful supercomputers of today having million-way parallelism [16].

It is important to note that supercomputing applications are different from those running on large capacity clusters, such as popular web services, i.e. enterprise computing. In supercomputing applications there is a high degree of connectivity between the calculations done on each computing core or node. On the other hand, capacity-focused clusters tend to deal with a large volume of data and requests that can be serviced mostly independently. The difference is tightly versus loosely coupled computing. Thus, the main concerns in supercomputing are different than those in enterprise computing.

2.1.1 Scaling

In principle, the scale of the available parallelism of a supercomputer can be increased simply by adding more and more nodes to a machine. While challenging in practice, the more fundamental issue is that the resulting parallelism would likely not be usable by most applications because the limiting factors often are the software and algorithms, not the hardware.

There are many obstacles to the efficient use of available parallelism. The first issue is lack of parallelism in the application – if the problem cannot be broken down into more independent tasks than there are processing units, then some will remain unused. If one can find enough independent work though, then other issues take over like contention over resources, computational intensity (number of computations per retrieved volume of data) and synchronisation cost. In addition, problems can arise with the hardware itself, these can cause complete failure, maybe prompting a restart to the latest saved checkpoint. Other hardware issues can be more subtle and reduce a component's performance without failing, thus breaking hardware homogeneity as-

sumptions a programmer may have had.

2.1.2 Synchronous algorithms

Many algorithms on supercomputers run in a bulk synchronous manner [3]. In this approach calculation steps alternate with synchronisation and data exchange steps repeatedly until some stopping criterion is satisfied. Any delays in the completion of synchronisation translate into increases in core idle time.

These delays happen when different computing units reach the end of an iteration at different times. This can be caused by load imbalance if the global problem does not divide evenly. Worse still, even a perfectly distributed problem would behave as though unbalanced if the processing units ran at different speeds.

This latter scenario is more likely to be encountered at large scale simply due to chance of sampling an under-performing core. Unfortunately, a large run would also experience higher cumulative loss of performance due system imbalance, since more cores would be waiting on the slow one.

2.2 Performance variability

Performance variability is the phenomenon where repeated runs of the same combination of application, input and hardware results in differing execution times. We will sometimes use the term “noise” as a shorthand for performance variability.

As the scale of supercomputers grows, synchronisation in applications is an increasingly important issue. The performance of individual cores, sockets and entire nodes, that are identical on paper, is in fact variable. Increasing the number of components that an application is run on increases the likelihood that performance variation will be encountered; this is an especially important issue when considering exascale [17, 18].

2.2.1 Types and sources of performance variability

In the area of system dependability and resilience there are specific terms that are used to differentiate the stages of a “problem” within the system: fault, error and failure. As described in [19], a fault creates one or more errors, and a failure occurs when an error affects the delivered service. This thesis is focused on *faults that cause performance*

errors. To simplify terminology, we will refer to these errors as performance variability and noise.

Current large-scale computing systems face significant performance variability. Different parts of the system can encounter delays ranging from brief OS interrupts to an overheated node to broken components. Applications are largely unaware of this and, in order to be efficient, they try to do as much work as possible between synchronisation or communication points. Thus the different parts of applications that use multiple components of the system can become widely out of sync.

The sources of performance variation are many: background tasks, OS tasks, memory hierarchy, chip manufacturing differences, power management, resource contention, network congestion, heat throttling, frequency boosting, multiple users, data caching protocols, automatic checkpointing and coding mistakes. Hardware itself can be heterogeneous within a single machine, for example if it contains both CPUs and GPUs. However, the listed sources can make homogeneous hardware exhibit performance variation and thus appear heterogeneous.

The listed performance variation sources can be classified as performance faults. In supercomputing environments, exceeding a job's time allocation and getting cancelled is as costly as a having a job crash, so the cost of performance faults goes beyond the immediate runtime delays. A more in depth discussion about other fault types can found in [20] and about HPC failures in [21].

Note that noise can have a wide range of manifestation patterns. Both in terms of duration – transient, intermittent or persistent – and in terms of pattern – uniformly affecting all nodes versus systematically affecting some nodes more than others. This determines what kind of mitigation strategy is likely to be effective.

2.2.1.1 Core level noise

Performance variation can exist within a single multi-core chip. Historically, Bowman [22] showed that systematic variations in the lithography process can have a large (30% decrease) effect on maximum clock frequency. Shrinking transistors and decreasing running voltage makes this problem worse, as does chip ageing. Balakrishnan et al. [23] show that multi-threaded workloads tend to suffer in scalability and predictability if there is performance asymmetry on a multicore chip. OS awareness of core-to-core variation can mitigate the problem for some workloads, but this would also require application level awareness of performance variations. Humenay et al. [24] propose reducing performance asymmetry by boosting slow cores; this can be effective

but increases the possibility of thermal throttling which then makes the performance asymmetry dynamic. Alternatively, the existing variation can be used as an opportunity to specialise for power efficiency [25, 26].

Other sources of performance differences between cores can be attributed to random OS noise [27], Non-Uniform Memory Access effects (for example if data needs to be accessed from a memory bank remote to the socket) and growing heterogeneity in computing systems [18].

2.2.1.2 Node level noise

Clock speed variation between nodes is due to a combination of manufacturing differences and energy management [28, 29]. Central Processing Unit (CPU) chips may run at similar speeds when they are allowed to use as much power as they desire, but once there is a power cap imposed in order to meet an energy saving goal, clock frequency differences emerge due to power inhomogeneity. Similarly, thermal design power together with vector instructions can make chip performance highly unpredictable [30].

2.2.1.3 Network level noise

There can be variance in communication performance due to network congestion [31]. This is a major concern for HPC users in an operational setting [32] and has been tackled repeatedly by network designers [33, 34]. The increase in mean latency as well as the latency variance have significant effect on application performance [35]. The negative effect of latency variation is especially pronounced when collective operations (e.g. message passing interface (MPI) barriers) are used because the slowest participant determines the execution time. Latency variance generally grows with the increase in network congestion level due to factors like contention at Network Interface Controllers (NICs). Additionally, congestion usually appears in multiple regions of a machine [36], so nodes allocated to one job may experience different levels of communication performance. Furthermore, network variablity also affects the performance of input and output since data communication may be routed to designated reading and writing nodes.

2.2.2 Impact of performance variability

The probability of encountering performance errors increases with the amount of parallelism utilised and the length of a run. Some application areas, for example weather

forecasting, are time critical so possible delays need to be understood.

In synchronous algorithms, the rate of progress is ultimately limited by the last core to reach synchronisation points. Assuming perfectly balanced load, the slowest running core will be the limiting factor. Since the chance of encountering noise increases with increased scale, scalability is limited. Moreover, frequent and global synchronisation itself is a limiting factor for scalability since it is an expensive operation and is sensitive to performance variation. Even non-global synchronisation like neighbourhood communications (e.g. nearest neighbour halo exchange) will have similar issues if faced with persistent performance heterogeneity at exascale [37]. A delay in one node eventually propagates to the entire system as neighbours begin to wait on their neighbours' neighbours and so on. The runtime variability (maximum runtime divided by minimum runtime) of production grade applications on the Theta supercomputer was measured to be between $1.18\times$ and $1.74\times$ and was attributed to network congestion [38].

Ferreira et al. [39] have quantified the effects of OS noise on the performance of applications running on thousands of nodes. Their results show that some applications accumulate up to 3 orders of magnitude more slowdown than the injected noise, i.e. 2.5% injected noise results in 15% to 1900% slowdown. One of the main reasons for this is the presence of blocking collective communications – effectively synchronisation in most use cases. Even though [39] focuses on OS interference, the principle should apply to other sources of noise in a supercomputer as well, for example periodic checkpointing. In some cases it is possible to reduce these effects by refactoring the application to use non-blocking collectives. However, the effectiveness of this approach depends on the characteristics of the noise encountered and the library implementation of the collectives [40].

2.2.3 Ways to reduce or mitigate performance variability

As long as application algorithms contain frequent, synchronised global communications and global barriers, they have a large weak point that can allow noise to degrade performance and scalability. Ideally, one would like to reduce or eliminate performance variability, but the methods that attempt this come with their own challenges.

Operating System (OS) jitter can be greatly reduced by using specialised kernels for the compute nodes which remove background delays [41] or constrain them to cores dedicated to OS use [42]. A lightweight kernel can be a drawback for applications that

have come to rely on a fully featured Linux kernel, but there are hybrid approaches that allow the two to exist side by side [43].

Specialising network hardware to HPC workloads can reduce the impact of network congestion on performance stability. For example, Cray Aries interconnect implements adaptive routing which computes four possible transfer paths for each packet and combines them with information about the current load along those paths to find the best tradeoff between route length and congestion [33]. As another example, the Slingshot interconnect employs congestion control which detects communication patterns that are known to cause congestion – like incasts, broadcasts, all-to-all – and throttles the offenders, thus preventing them from obstructing traffic between nodes of another application [34].

Advances in hardware often come with unforeseen issues, including performance variability. Recent examples include Intel Xeon Phi processors which were observed to have significant performance variability at the core, tile and node level [44] and Intel Xeon Platinum 8160 processors which suffered from infrequent but consistent slowdowns in certain linear algebra benchmarks [45]. These problems can be partially or fully alleviated, but this requires extensive investigative and debugging effort and, moreover, there is no guarantee that future systems will not introduce new performance variability sources.

Another approach is to over-provision cores or nodes for an application and use the extra resources to mitigate noise. For this method to be effective, the amount of machine resources set aside have to be less than the amount of wasted time that is prevented. Over-provision has been used with synchronous stochastic gradient descent [46] in computations where a central node needs to receive an update from N worker nodes. A computation is initiated on $N+b$ nodes, the first N results to arrive are recorded and the results from the slowest b nodes are dropped. A more general solution may be based on detecting abnormally performing cores or nodes and migrating work away from them automatically, but it would be difficult to do so optimally based only on external observables like performance counters.

Yet another option is to introduce communication and computation overlap into an application. This helps absorb random noise [47], given a sufficient percentage of overlap. However, it is limited in the face of persistently slow components. In addition, there normally is a significant amount of work involved in rewriting code to add or increase the overlap, if it is possible at all.

Finally, one can use load balancing to adapt an application's work distribution to

the available compute resources. In general, static methods, where the load is distributed once at the start of a run, are not able to deal with a noisy environment. Dynamic methods are better suited due to their ability to adapt work distribution at runtime. However, this normally comes with more complexity both in implementation and tradeoffs between perfect balance and the cost of achieving it (e.g. moving work between nodes). An effective example of this approach is Charm++ [48]. In this framework, applications are expressed in terms of mobile work and data units so dynamic load balancing can be applied with great flexibility, but the user must first incur the cost of extensive rewriting of the target application.

The methods listed here can be helpful in certain situations, but it is difficult to bring them all together and the underlying issues are likely to continue into the future. It had been projected that faults will be more frequent at exascale [18], to the point of occurring on the order of minutes. However, through extensive efforts like those which have been applied to removing OS jitter [41, 42] and increased spending on premium hardware (see CPU binning [6]), significant gains have been made towards reliably performant systems. Nevertheless, performance variability continues to be an issue in both shared (as seen in Figure 1.1) and distributed memory systems [38]. We will therefore focus on an alternative to frequently synchronising algorithms, i.e. algorithms which can tolerate performance variation, rather than try to eliminate noise.

2.3 Asynchronous algorithms – an alternative

Given the efficiency limitations of large scale synchronisation, it is attractive to consider asynchronous algorithms wherever possible. These are normally iterative convergent algorithms that do not rely on strict synchronous execution. Asynchronous algorithms can progress using "stale" values, so one worker (e.g. a core, CPU or node) would not have to wait on another that may have stalled, but instead use the most recent value from the stalled worker.

However, using stale values replaces performance variability with "progress variability", i.e. some parts of the problem space will have progressed towards the solution more than others. In general, the time to solution is a function of iteration rate and convergence rate. The iteration rate tends to be higher for asynchronous algorithms since time is not wasted waiting on results from others workers. The convergence rate can be lower for asynchronous algorithms due to the use of stale values for updates and too much staleness can result in non-convergence [49].

2.3.1 Basics of asynchronous algorithms

Some of the earliest work on asynchronous algorithms is Chazan and Miranker's seminal paper on chaotic relaxation [7] in 1969. It provides the theoretical conditions of convergence for a class of matrices to solve systems of linear equations, including Jacobi's algorithm (see Section 2.3.2).

The key differentiating feature between synchronous and asynchronous algorithms is that workers in asynchronous algorithms have the ability to carry on working (if they do not terminate) with missing, incomplete or "stale" data. The workers can operate independently up to a point, but communications with other processes lead to better or quicker results.

A synchronous algorithm is susceptible to frequent stalls due to dependence on the results between parallel workers. Even if workload was balanced perfectly, computing cores may run at different speeds either due to heterogeneous architecture or noise in the machine. Additionally, if a process fails, processes that depend on the failed one's results must terminate or wait for a restart. The restart may be very expensive if the failed process cannot be restarted in isolation, i.e. if it needs past states from other processes in order to reconstruct its state at the point of failure.

Asynchronous algorithms have the potential to do away with both of the above issues. Processes can run freely and use the latest available information from other processes without waiting. This allows fast processes to carry on and potentially pull others along by providing them with better information. Asynchrony also implies fault tolerance because, if processes can work with missing data, a process failing should not be a problem or be easily handled. A downside of asynchrony is the increased complexity of theoretical analysis and providing convergence guarantees.

Another issue with asynchronous algorithms is the nature of their output. An asynchronous algorithms will take a different set of steps towards the solution each time it is run. This means that the results, while converging to the same value, will not be reproducible without a record of the steps taken. For certain uses, such as safety-critical engineering simulations, this will likely prove to be a significant barrier for adoption of the methods, but, if it means performance or fault tolerance gains otherwise unachievable, users may be willing to reconsider their requirements. Additionally, synchronous algorithms can have issues with exact reproducibility as well due to the limitations of floating point calculations, for instance when changing the decomposition of a problem or the number of cores it is run on. It is possible to combat these issues in synchronous

algorithms by using specialised arithmetic techniques which are independent of the order of operations [50] but there is a performance cost to pay.

In 1976 Kung [51] defined the notion of semi-synchronised algorithms – a middle ground between synchronous and asynchronous algorithms – where there is a limit on the relative progress that threads can have between them. They allow using stale values up to a predetermined degree of staleness. If the limit is never reached, semi-synchronous algorithms can benefit from nearly all of the advantages of asynchronous algorithms. However, if the staleness limit is reached, workers that that have received the excessively stale values begin to stall until sufficiently fresh values are available. A single worker that consistently falls behind can cause other workers to fall behind, eventually leading to the algorithm effectively behaving like a synchronous one. See Figure 2.1 for an illustration of the difference between synchronous, semi-synchronous and asynchronous algorithms.

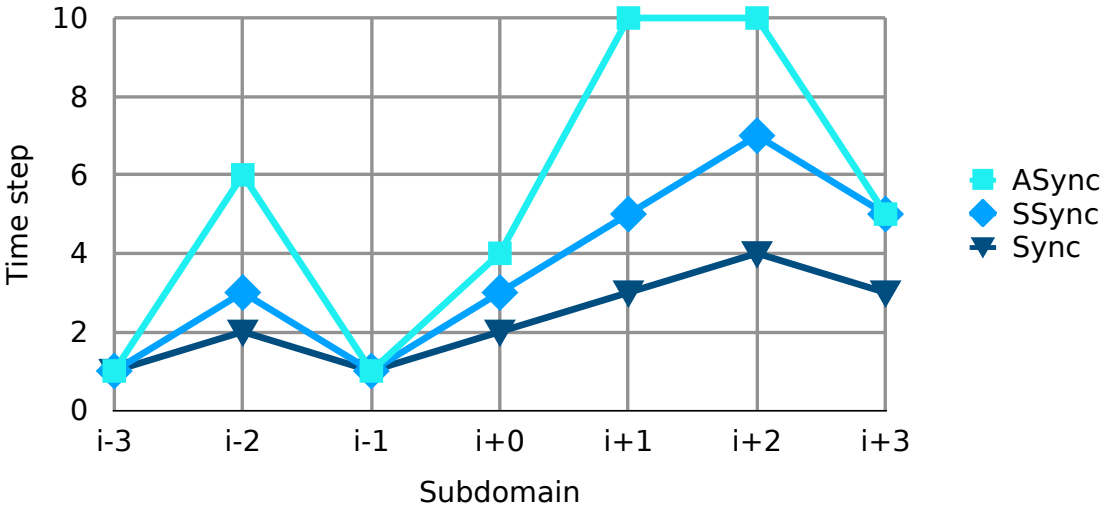


Figure 2.1: Illustration of the difference between three parallel algorithms with different types of synchronisation. The algorithms are iterative, 1-dimensional and each sub-domain computation depends only on nearest neighbour communication. Subdomains i-3 and i-1 take a long time to process. The synchronous algorithm cannot progress without up-to-date values from its neighbours. The semi-synchronous one imposes a staleness limit of 2, and beyond that it cannot progress. The asynchronous one allows arbitrarily large gaps in progress between neighbouring points.

2.3.2 Detailed example of asynchrony

Let us examine a particular algorithm in detail, Jacobi's algorithm. Its general form is used to solve the system of linear equations $Ax = b$ where the matrix A and vector b are known. A can be expressed as the sum of its lower triangular, diagonal and upper triangular parts $A = L + D + U$ and used to rearrange the original equation as

$$x = D^{-1}(b - (L+U)x). \tag{2.1}$$

In this form x can be iteratively approximated, starting from a random vector, by using the previous version k to compute a new version $k+1$, i.e.

$$x^{(k+1)} = D^{-1}(b - (L+U)x^{(k)}). \tag{2.2}$$

Eventually the approximation stabilises (given some convergence criteria) and the system of equations is solved. We will focus on the diffusion problem $\nabla^2 f = 0$ where f describes some continuous quantity measured over a 2D domain with Dirichlet boundary conditions. The problem is discretised using finite differences which allows a simple implementation of the algorithm as a stencil pattern where x represents the discretised solution space and is updated using nearest neighbours in 2D space via

$$x_{i,j}^{k+1} = 0.25(x_{i+1,j}^{k} + x_{i-1,j}^{k} + x_{i,j+1}^{k} + x_{i,j-1}^{k}) \tag{2.3}$$

where i and j denote coordinates in the x and y dimensions in the discretisation of the problem domain. Figure 2.2 illustrates this pattern. Stencil algorithms are commonly parallelised by decomposing the domain and distributing it to multiple workers that can then update their subdomains simultaneously. To update the elements on the edges of domains, workers need to retrieve values from the edges of neighbouring domains which are referred to as "halos".

A basic implementation is shown in Algorithm 1. The steps are executed on multiple workers in parallel, which need to synchronise at every iteration. The requirement can be relaxed by only synchronising between workers that exchange halos. Despite this simplification, all workers are dependent on the progress of all other workers via some number of hops.

Jacobi's algorithm can be run asynchronously, provided that the condition that the spectral radius $\rho(|-D^{-1}(L+U)|) < 1$ is met [7], and the diffusion problem does indeed do this. Algorithm 2 shows an implementation of asynchronous Jacobi. Note that the conceptual changes from synchronous Jacobi are minimal, only the removal of the

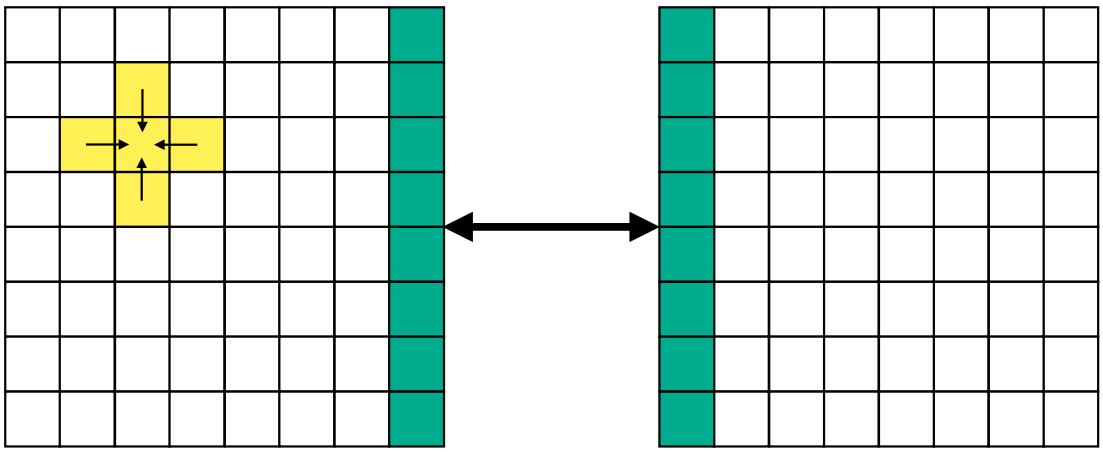


Figure 2.2: Illustration of the Jacobi algorithm on a 2D diffusion problem distributed between two workers using domain decomposition. The yellow cross shape shows an update to a cell using the values of neighbouring cells. The green shaded areas are halos that need to be sent from one neighbour to the other in order to compute updates to edge elements.

synchronisation point at the end of each iteration. In an actual implementation there are more considerations, like ensuring safety against data races and facilitating an irregular communication pattern with multiple, on-demand message exchanges. Regardless, the principal change remains simple.

The middle ground is found in semi-synchronous algorithms (see Algorithm 3). Here the explicit synchronisation point is removed again, but its place has been taken by a check for halo staleness. If a received halo is too stale, the worker that depends on it waits until the other worker produces a sufficiently up-to-date halo. The threshold can be tuned to the required level of progress equality, and in fact covers both the synchronous (threshold of 1) and asynchronous (threshold of infinity) cases.

An important distinction to make is that we are not considering task based parallelism. While these algorithms are also called asynchronous, the word “asynchrony” here refers to replacing global synchronisation with point-to-point synchronisation to satisfy data dependencies. For example (see Algorithm 4), in C++ it is possible to execute functions *asynchronously*. The call returns immediately, allowing other work to be done, and gives the user a *future*. Once the result of the function becomes essential to further progress, one has to *wait* on the *future*, which is a blocking call. The function may have already finished or it may only start when the *wait* is issued – this is the sense in which the work is asynchronous.

```
grid, newGrid ← initial values;
while not converged do
    GetHalos();
    newGrid ← UpdateGrid(grid);
    grid ← newGrid;
    SynchroniseWithNeighbours();
end
```

Algorithm 1: Synchronous Jacobi.

```
grid, newGrid ← initial values;
while not converged do
    GetHalos();
    newGrid ← UpdateGrid(grid);
    grid ← newGrid;
end
```

Algorithm 2: Asynchronous Jacobi.

```
Data: MaxStaleness
1 grid, newGrid ← initial values;
2 while not converged do
3     GetHalos();
4     while GetHaloStaleness() > MaxStaleness do
5         GetHalos();
6     end
7     newGrid ← UpdateGrid(grid);
8     grid ← newGrid;
9 end
```

Algorithm 3: Semi-synchronous Jacobi.

```
do some work;
future ← async(EstimatePi());
do some other work;
result ← wait(future);
do some further work;
```

Algorithm 4: Asynchrony in the task-based parallelism sense.

2.3.3 The application of asynchronous algorithms

Research on the convergence of asynchronous algorithms, mainly theoretical, continued until 2000, when interest started trending towards their fault tolerance aspects in the context of grid computing [52]. While there have been successful applications, generally asynchronous algorithms did not gain widespread adoption because synchronous algorithms scaled sufficiently well up to the required and available computational scale at the time.

As the scale of supercomputers is increasing, however, it is worth reconsidering asynchronous algorithms. Indeed, the research community has retained interest in asynchronous algorithms and their application has expanded to multiple areas; examples include:

- relaxation methods for linear systems of equations [7, 9]
- finite difference solvers of PDEs [11, 53]
- adaptive mesh refinement methods [54, 55]
- Schwarz methods [8]
- stochastic gradient descend (SGD) [56, 57]
- genetic algorithms within computational fluid dynamics [58]
- extracting performance measures from performance models [59]
- dynamic programming and network flow problems [60]
- relaxed memory consistency model [61, 62]
- graph processing [63]

A large portion of these algorithms are iterative of the form $x := f(x)$. The common property is an algorithm that proceeds by improving on a previous solution. This makes intuitive sense: if an algorithm proceeds to the correct solution through a "space" of approximations to the solution in multiple steps, then it is not unreasonable to expect that there is scope for deviations from the intermediate steps of the path.

2.3.4 Progress variability in asynchronous algorithms and ways to address it

One of the key reasons to consider asynchronous algorithms is for their apparent resistance to performance variation of hardware. However, noise results in "progress variation" which can be detrimental to performance as well.

The pattern of noise determines the profile of the variation of progress. Uniformly random noise would not cause a large amount of progress variation. The slack afforded by asynchrony allows some parts of the problem domain to fall behind for a time and later catch up as other parts are affected. This is the best case scenario for asynchronous algorithms and largely for semi-synchronous algorithms as well, given that the staleness bound is not too small. Conversely, a synchronous algorithm would have to wait on each instance of noise. However noise is not guaranteed to fall uniformly and even small patterns accumulate. Moreover, persistent sources of noise like chip manufacturing differences would quickly cause a large degree of progress variation for asynchronous algorithms.

A 1991 survey of the work done on asynchronous iterative algorithms done by Bertsekas et al. [64] classify them into three categories based on how much asynchronism the algorithms can tolerate. These are (i) totally asynchronous, (ii) ones that require just some bound on how much processes can fall behind and (iii) ones that require a small bound on how much processes can fall behind. The existence of these distinctions shows that staleness of updates must be considered when working with asynchronous algorithms.

As the application of asynchronous algorithms spread to more areas, the potential issues with staleness pointed to by theory became more apparent and explicit in practice. There have been multiple different approaches to addressing these issues.

Total asynchrony

The simplest approach is to ignore progress variability and simply let an application run fully asynchronously.

Bethune et al. [65] have examined an asynchronous Jacobi solver in the context of large scale supercomputing. Running the algorithm at large scale (up to 32768 cores) revealed important practical details that purely theoretical work does not fully address. Overall, the asynchronous implementation needs to complete more iterations to converge but due to an increased iteration completion rate, usually the runtime is

lower. Also, a case is documented where a single core running at half speed doubled the runtime of a 32k core synchronous run, but had no effect on the asynchronous runs.

However, it was observed that there is a wide spread of completed number of iterations per core, despite the machine having homogeneous chips. At 32k cores in one of the asynchronous implementations a small subset of processes experienced very poor communication performance, leading to a large variability in completed iterations and a doubling in runtime and average iteration count. It is possible that the slow processes held back the application from reaching its convergence criterion.

A degree of variance in iterations appears tolerable and even beneficial, but at some point it starts to degrade performance and convergence.

Total asynchrony with modified updates

In some situations it is possible to retain total asynchrony but limit its negative impacts by treating stale values with extra care.

An interesting method that is related to asynchronous algorithms is Hogwild [66], and it handles stale values by avoiding them in the first place. The basic algorithm is stochastic gradient descent (SGD). Multiple threads are used in order to spread the work of updating the new optimum value. No locks are used when updating shared variables because they are sparse and the basic algorithm is resistant to noise. As a result overwriting of values is rare and can be ignored or dealt with when it does happen. It is a good solution for a subset of applications that perform sparse updates.

When sparsity cannot be guaranteed, one has to accept that there will be significant differences in iteration rates and instead try to manage the negative effects of asynchrony using various algorithmic corrections. For asynchronous SGD examples include tuning algorithmic momentum [67] based on the degree of asynchrony, skipping updates that would direct away from a projected solution [56] or compensating for delayed gradients caused by calculating gradient updates using a stale snapshot of global state [68]. A limitation of these approaches is a lack of generalisation to other applications.

Donzis et al. [53] have used a statistical framework (since arrival of new data can be viewed as a random process) to analyse finite difference solvers for PDEs. They show that asynchronous finite difference solvers for PDEs always drop to first-order accuracy and that the error is proportional to the number of cores and mean delay. They then use this framework to show how higher-order schemes can be derived, ones that are robust to asynchrony on a mathematical level. This is done by using larger stencils that give

room to cancel out problematic terms. Analysis of the schemes shows that they are not stable in theory, but in practice they retain stability, so more work is required to find tighter bounds on the stability region.

Amitai et al. have published work on asynchronous schemes used to solve PDEs (in particular, the heat equation) using finite difference schemes. In [69] they show that a completely asynchronous solver is only reliable for steady state problems, i.e. ones where the solution converges to a stable state. Instead they propose a corrected asynchronous scheme with more accurate results for intermediate values, but with a loss of efficiency. This scheme requires enforcing of bounds on a value that is a combination of time step size, grid resolution and, crucially, the amount of asynchrony. The latter further reduces the practicality of this scheme in the presence of continuous noise.

Solutions like these can be very effective, but they tend to be specific to a set of applications.

Limited asynchrony with a hard bound

Any asynchronous algorithm can be made semi-synchronous if staleness cannot be circumvented otherwise.

A semi-synchronous algorithm for machine learning applications has emerged under the name "bounded staleness" [70, 71, 72]. In this approach there is a hard limit on how out-of-date values can be before a worker has to wait for fresher ones. Some use cases are: topic modelling, PageRank, collaborative filtering, sparse regression, conjugate gradient and all-pairs shortest path. These algorithms converge iteratively, some of them do so by doing multiple passes over subsets of data and then aggregating results. It was found that allowing a few iterations of staleness improves performance, but too much slows convergence.

Bounded staleness has been implemented as a memory consistency model. Vora et al. [61] used it in the context of distributed shared memory. They set out to balance the use of stale values and tolerance to communication latency; this is done by imposing a hard upper limit to staleness and supplementing it with a best effort refresh policy where additional threads preemptively refresh stale values. Using vertex-centric graph algorithms for evaluation, they show improvement of $4.2\times$ over synchronous implementations and $2.27\times$ over a purely asynchronous implementation. Lee et al. [62] evaluated a bounded staleness implementation on a hardware level. Their work showed $1.33\times$ speedup over a baseline asynchronous implementation on machine learning applications.

While this is the best method to limit staleness from occurring, it also loses some of the performance benefits of total asynchrony. This is especially an issue in the presence of long-lasting performance variability, because delays propagate and a semi-synchronous algorithm becomes effectively synchronous when all workers reach their staleness limit.

Limited asynchrony with a soft bound

The best generally applicable solution would limit staleness but not lose the benefits of asynchrony. This suggests semi-synchrony with a "soft" staleness bound.

An example can be found when running asynchronous algorithms on GPUs. In this setting they face systematic biases in updates due to patterns in GPU thread scheduling [73]. This can lead to amplification of variations in convergence between asynchronous blocks. The problem can be tackled by managing the order of execution of thread blocks [74]. This method effectively aims to reduce the staleness of the values used for new updates. It can increase convergence rate, but it is not clear how one would apply it to multi-GPU setups.

A more passive approach has been applied to large scale deep learning [57]. The core asynchronous computation of the application is tuning of model parameters. However, the application is not entirely asynchronous. Workers are organised into groups; communication within groups is synchronous but communication across groups is done asynchronously through multiple servers that store and update model parameters. This hybrid asynchronous-synchronous scheme offers a way to balance statistical and hardware efficiency by tweaking the sizes and the number of synchronous groups, thus controlling the scale of asynchrony in the system. A similar method has been applied to linear solvers by combining synchronous Krylov subspace methods with asynchronous Jacobi [75]. The disadvantage of these approaches is that the balance is chosen at the start of a run and dealing with performance variation changes at runtime is difficult and costly.

In order to deal with performance variation that is unknown before runtime, one has to consider dynamic load balancing. This area is an active field of research, however the majority of techniques are developed for, and applied to, synchronous algorithms and so may not transition well to asynchronous algorithms or require significant changes to the techniques. For example, work stealing is a popular and scalable method [76]. Workers process a local queue of tasks and when they run out, more work is stolen from work queues of other workers. In this form it cannot be applied

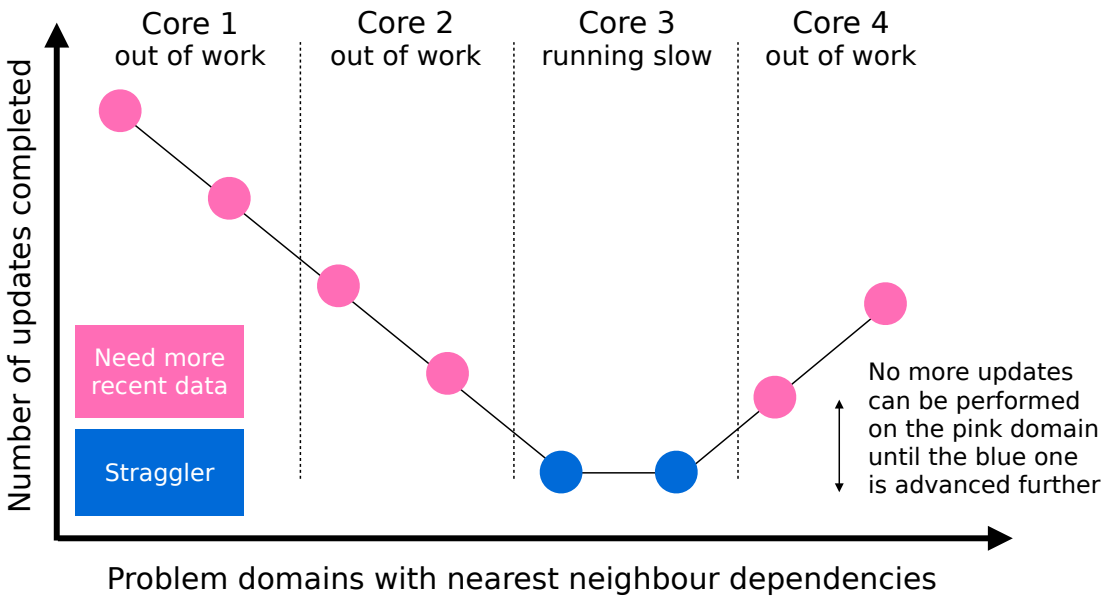


Figure 2.3: Illustration of work stealing when applied to a semi-synchronous algorithm. The global problem domain is divided among cores along the dashed lines.

to asynchronous algorithms because threads in principle never run out of work and always appear busy, so they would just use continuously more stale values.

Work stealing can be applied to semi-synchronous algorithms, where a maximum staleness bound is enforced so the amount of work available per worker does have a limit. However, our experiments (see Section 4.2.4) showed that the method does not work well in the semi-synchronous case because the system soon reaches a state where there are many starved workers and not much work to steal. Straggler workers gradually increase overall staleness, which depletes available work (see Figure 2.3), and stealing work at random has a lower chance of stealing from a slow core than a normal cores, so the depletion is accelerated further. Reaching this point does not necessarily mean that the application is finished because it likely needs to continue iterating until the desired termination criterion is met. New work is made available only when the stragglers make progress, this releases a "wave" of new work that travels across the problem domain. The new work is consumed as soon as it is made available and as a result the system essentially behaves as a semi-synchronous code would do without work stealing. The lack of awareness of the properties of asynchronous algorithms prevents standard work stealing being an efficient load balancing technique. Hence we are focusing on techniques that have been shown to be applicable to asynchronous algorithms, which is a key criterion for us.

A good example of load balancing an asynchronous algorithm in distributed mem-

ory can be found in [77] where Bahi et al. apply balancing to a 1D stencil application. This algorithm sends parts of the working array from one worker to a less loaded neighbour, where the specific decision boundaries are based on the theoretical analysis of the load balancing problem in [60]. They found that significant performance gains could be achieved by balancing the iteration rate between components in a grid computing context. Even bigger gains could be gained by using the residual as a load estimator. While showing excellent speedup (up to $5\times$ in the grid computing setting), the proposed algorithm is presented in 1 dimension only where redistributing workload by changing domain boundaries is trivial. An extension to multiple dimensions would be difficult to design and implement.

We believe the load balancing approach the most promising for dealing with staleness while retaining the benefits of asynchrony, however the work done in this area is limited.

2.4 Summary

Growing demand for supercomputing capability has lead to ever more parallel components in modern systems. The advances in technology and complexity have brought with them many sources of performance variability – at the core, node and network level. This is a significant barrier for many applications to make efficient use of the available compute resources, due to widely employed synchronous algorithms. Performance variability is unlikely to be eliminated, so methods that tolerate it are attractive.

A promising approach is to exploit a class of algorithms known as asynchronous algorithms in which workers can continue computation using stale input from other parallel workers. There is substantial interest in parallel asynchronous algorithms in the literature. It spans at least 6 decades and a spread of application areas. A commonality in methods is the iterative improvement upon a past state. Early work is mostly theoretical but in recent years there have been more practical evaluations. The methods have been shown effective in multiple settings and usually perform better than synchronous counterparts.

A frequent concern for researchers in this field is dealing with stale values; performance variation is still a factor for asynchronous algorithms – we refer to the resulting issue as progress variation. Fluctuations in progress across workers at best delays convergence to the result and at worst invalidates answers. It is possible to apply algorithmic or mathematical corrections to deal with stale values in some cases or limit

the maximum staleness by using a semi-synchronous algorithm instead. However, for more general applicability and to preserve the performance benefits of asynchrony, a dynamic load balancing approach should be considered. The existing work in this area provides a good starting point, but there is scope for improvement with emphasis on reduction of progress variation and wider applicability. In the following Chapter we will outline our approach in designing a load balancing method for asynchronous iterative convergent algorithms.

Chapter 3

Designing a load balancing scheme

Given the limited applicability of past solutions to performance variability and resulting staleness in asynchronous algorithms, we are motivated to seek an alternative approach via dynamic load balancing. In this Chapter we present our approach in which we exploit the unique property of asynchronous algorithms to tolerating staleness. This allows us to load balance in a different way – over time as opposed to instantaneously. Indeed, we do not attempt to *equalise* the iteration rate (or any other load metric), but rather vary it to keep a different metric – progress variation – bounded. Balancing is done over time by effectively swapping iteration rates of different problem subdomains. Update rates of problem subdomains keep changing, but the difference in number of updates between subdomains is bounded.

3.1 Simulator

As discussed in Section 2.3.4, an asynchronous algorithm can tolerate uniformly distributed random instances of noise, as these effectively cancel each other out over time. Therefore we focus on larger and more localised noise events, or performance variability that follows patterns and hence accumulates over time. Such noise is generated by artefacts like chip manufacturing differences and poor workload distribution. We want to tackle these larger noise events by using a similar principle to the one that cancels out small random noise. Our approach aims to take advantage of the unique property of asynchronous algorithms to tolerate temporary staleness. This leads to the development of a system where iteration rates oscillate.

We investigated possible load balancing techniques using a simulation framework. Here we describe the design and implementation of the simulator and how it was used

to develop load balancing algorithms. We are not aware of existing simulators applicable to asynchronous algorithms, so we designed our own.

Simulating an asynchronous algorithm along with a load balancing scheme can be viewed as modelling the problem that we are trying to solve. Alternatively a mathematical model of the problem could be developed, but, due to its non-deterministic nature, this approach was considered unfeasible for this work. The complexity comes from asynchrony (there are no convenient synchronisation points where a consistent state of the system could be described) and from coupling of the application and the load balancer. Using a simulator instead allows us to specify the processes in as much detail as necessary.

Using the simulator we were able to rapidly test various methods of load balancing and find the most promising techniques as well as identify edge cases where the load balancing might fail.

3.1.1 Simulator design and implementation

We assume that the performance variability persists for a number of iterations of the asynchronous algorithms, leading to a gradual divergence of progress (number of completed iterations) between problem subdomains that are being updated asynchronously. This assumption allows us to approximate the progress of a work item just by using iteration rate, which is derived from some speed metric of a processor and the amount of work to be done by the processor. The simulator is still valid if the performance variability does not persist for the entire run of an application. Sections where noise appears or disappears can be viewed as a chain of separate runs. If the simulation is valid within each piece of the chain, it is valid in the whole chain.

Given these assumptions, the problem can be modelled and simulated computationally. The simulator is implemented using the Python programming language. We simulate the problem using a time stepping approach.

We start by creating a set of processing elements (PEs). These generally correspond to physical elements, like CPU cores. Each PE has a speed assigned to it akin to clock frequency in units of computational power (e.g. cycles) per second. The PEs can be customised to have random deviations from the base frequency to simulate general manufacturing variability or assigned a slowdown factor to specific PEs to simulate faulty components.

Next we create a set of work items (WIs) that represent the global problem domain

broken down into subdomains which can be updated in parallel. Each WI is given the same computational cost to complete one update on it. The WIs are then given a topology of communications, e.g. the WIs could be placed in a 2D or 3D grid, or a torus. Each communication link has a cost associated with it (communication happens between iterations) and this is added to the computational cost of each WI. As a result, the total cost of updating different WIs may be different if they have an unequal number of communication links (e.g. "edge" versus "middle" subdomains). Note that there is no actual solver present and WIs merely represent workload, since the goal is to evaluate load balancing in abstract.

At the start of a simulation each PE is given one or more work items. The work items can be assigned to available PEs either according to the topology of the problem, in a round robin fashion or randomly (but balanced in count).

The simulator steps through time and increases the iteration counter on each WI by a number calculated based on the speed of the PE, the total computation cost of the set of WIs belonging to the PE and the size of the time step.

Periodically a load balancer is activated, which redistributes the WIs across PEs according to some policy. Since we assume that the iteration rate of work items is stable, the load balancing is the only factor that changes the rate. Thus the time between load balancing events is also the length of a time step in the simulation. Furthermore, WIs are self-contained and moving them between PEs is therefore trivial in the simulator.

Simulation steps

(1) Time step. Progress all work items by one time step of length $stepSize$ which is the period between load redistribution events. A time step has units of seconds. Thus multiplying the time step by the computational power of a PE yields how much work can be done in the time step. In our model these are the same units as the computational cost required to update a WI. We can then calculate how many updates each WI is to be progressed in the corresponding time step.

The natural way to apply updates to WIs is round robin. For each PE:

1. get the number of cycles available $cycles = stepSize \cdot PEspeed$
2. get the computation and communication cost $workWI_i$ of each WI that belongs to this PE
3. iterate through WIs repeatedly, incrementing the update counter of WI_i by 1 if $cycles \geq workWI_i$. Decrease $cycles$ by $workWI_i$ after each increment

This method is faithful to real world applications since WI updates are normally executed in totality before moving from one WI on to the next. However, unless the available cycles divide evenly among the WIs, some additional mechanism is required to carry over unused cycles and an identifier of the last updated WI to the next simulation step. Ignoring the remainder cycles would lead to systematic errors in the simulation, e.g. the last WI on the list could always be left an update short. Therefore, this approach would complicate the simulator by requiring to keep memory between time steps in the simulator.

One can improve the time stepper using what we call "equal progress simplification". For each PE:

1. get the number of cycles available $cycles = stepSize \cdot PEspeed$
2. get $totalWork$, the sum of the computation and communication cost $workWI_i$ of all WIs that belong to this PE
3. increment each work item's updates by $updatesPerWI = \frac{cycles}{totalWork}$ (which does not have to be a whole number)

By allowing fractional updates in the simulator, this method simplifies the implementation significantly. No history needs to be kept between time steps and the step does not change if WIs are moved between PEs as a result of load balancing. It also matches how work would be progressed in the round robin approach asymptotically. The more iterations are performed in a time step, the smaller the relative error is.

(2) Redistribute load. Apply the load balancing policy that was chosen at the start of the simulator run. It moves WIs between PEs according to progress metrics of the WIs. This is done using local in time knowledge (however, a history could be kept) and without reference to the computation cost of each WI or speed of each PE (this information is readily available in the simulator but not in the real world, where it has to be estimated).

There are two simplifications at this step. It is assumed that moving work items is free and immediate. In a shared memory implementation(which is the primary target of the simulator) this would often be close to reality, given the cost of updating the WIs themselves. Regardless, the main goal of the simulator is to explore a variety of load balancing policies quickly and to evaluate their correctness, not their overheads in detail, as these would be highly dependent on hardware and the problem that is being solved.

Number of PEs	36
PE arrangement	6×6
PE cycles	4e6 per second
Number of WIs	36
WI arrangement	6×6
Computation cost of WI	95 cycles per iteration
Communication cost of WI	5 cycles per neighbour per iteration
Simulation length	5 seconds
Load balancing frequency	100 times per second

Table 3.1: Simulation settings.

3.1.2 Designing the load balancing algorithm

Following the implementation of the simulator, we used it to develop and evaluate various methods for load balancing. The goal of the balancing is to minimise progress *spread*: the difference between the maximum and minimum number of iterations completed on a work item across all PEs. This quantity represents progress imbalance. Additionally, the spread should remain stable and not grow with the length of the simulation.

An alternative metric to consider is *neighbour spread*. In many applications work items only directly interact with a few neighbouring work items. Thus one could define the target metric as the maximum difference between the number of iterations between any two interacting work items. However, global spread is an upper bound on neighbour spread and is easier to use.

In the simulator we have complete knowledge of the workload and the machine. The optimal load balancing decision could be found (or close to it) with the information available. However, there is more uncertainty in the real world and moving of work items are not isolated events. These secondary effects could be modelled as well, but then the decision space grows enormously and finding the optimal solution becomes too expensive and complicated. Therefore, all current load balancing policies are heuristics.

We now present a series of balancing policies that we evaluated. The data are simulated runs (5 simulated seconds in length each) of a 2D iterative solver workload with nearest-neighbour interactions and non-periodic boundaries. The settings of the

simulation are shown in Table 3.1.

The results are presented in Figures 3.1 to 3.9. In each graph the top row shows how the number of performed iterations on each problem domain evolve over time; there is one line for each WI. The bottom row shows how the difference between the most and least updated domains (iteration spread) progresses. The letter "k" indicates thousands of iterations. The vertical scale is kept constant where the range of values permits it. Each set of experiments contains 3 kinds of runs:

PE variation Variation is present in PEs only: the available number of cycles for each PE is multiplied by a random number between 0.9 and 1.0 at the beginning of the simulation. Communication between WIs is turned off so that all WIs take the same amount of work to process one iteration.

WI variation PEs are all identical, but WIs are allowed to communicate, leading to a workload imbalance because WIs with more neighbours take more work to process one iteration.

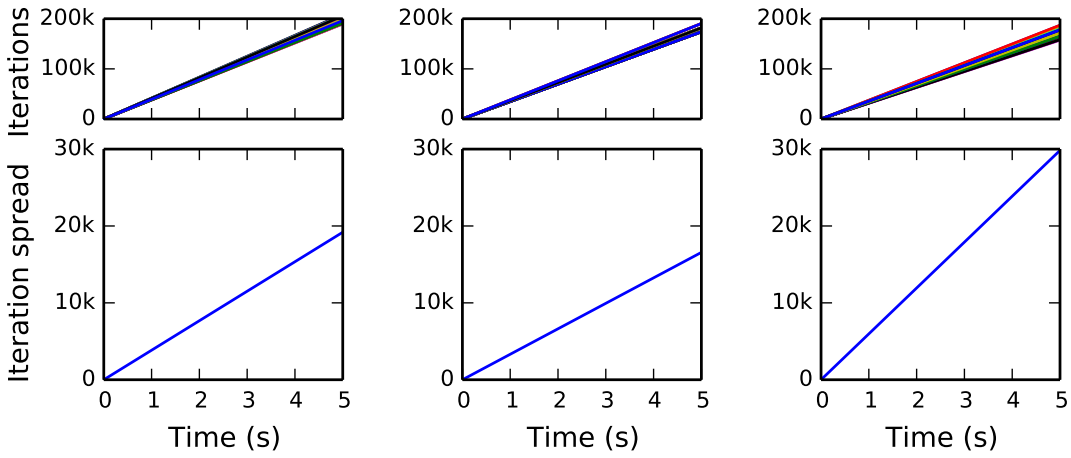
PE and WI variation Variation is present in both PEs and WIs. The available number of cycles for each PE is multiplied by a random number between 0.9 and 1.0 at the beginning of the simulation and WIs are allowed to communicate, leading to a workload imbalance.

The PE noise factor of 0.9 was chosen because it is similar to the workload imbalance between WIs, thus making comparison between the two types of variation straightforward.

There is a degree of randomness in the results, for example due to the assignment of PE speed variations. The plots shown here are representative examples of each balancing method. The same seed for the random number generator is used in each case.

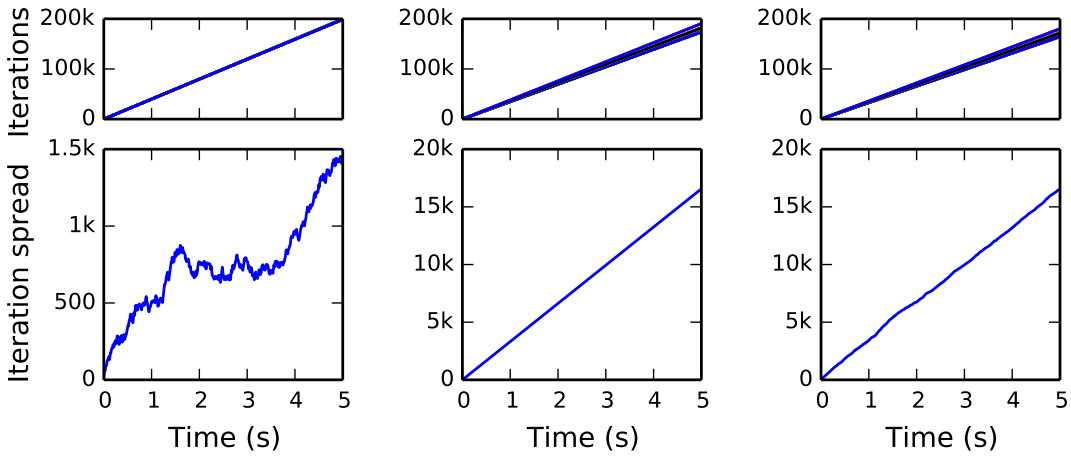
1 – No balancing

No balancing whatsoever takes place; this is the baseline case. Figure 3.1 shows the output of the simulation and it can be seen that the relative progress between WIs diverges over time. This can be seen more clearly in the bottom row of plots. The spread in the combined noise case is less than the sum of the individual cases as some of the workload and processing speed differences balance each other out.



(a) PE variation. **(b)** WI variation. **(c)** PE and WI variation.

Figure 3.1: No balancing.

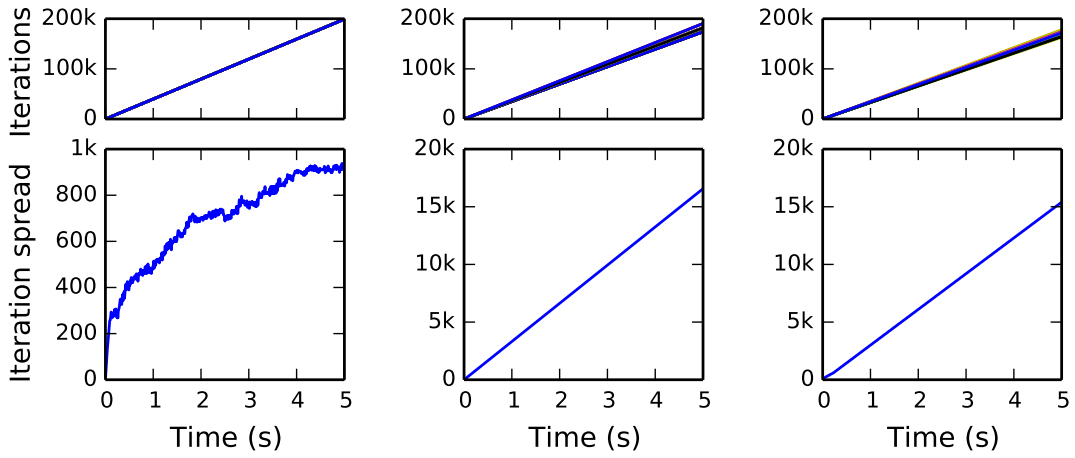


(a) PE variation. **(b)** WI variation. **(c)** PE and WI variation.

Figure 3.2: Randomisation policy.

2 – Random load balancing

Randomly assign WIs to PEs while keeping the number of WIs per PE constant. This policy significantly reduces progress variation due to PE noise but does not affect variation due to WI noise (see Figure 3.2). Randomly moving WIs between PEs of different speeds spreads out the unevenness, however it still builds up over time. Moving WIs has no significant effect on spread if the WIs themselves are the cause of the variability.



(a) PE variation. **(b)** WI variation. **(c)** PE and WI variation.

Figure 3.3: Swap min and max policy.

3 – Swap min and max

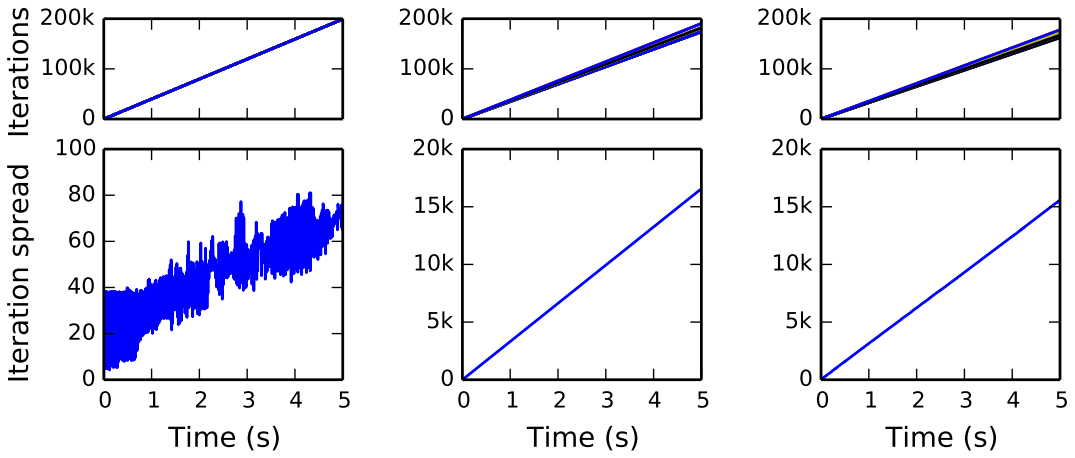
Swap the pair of least and the most updated WIs between PEs. This policy is generally useful only if there is a speed difference between PEs, as shown in Figure 3.3. It performs better than the random policy and in the PE noise case it can reach a stabilised level of iteration spread.

4 – Swap all min and max

Swap all pairs of least and most updated WIs between PEs. This method does very well when only PE speeds are different, but performs similarly to the methods presented so far in other cases (see Figure 3.4). In policy 3 multiple balancing periods need to pass before all WIs are affected by balancing, thus there is more time for spread to grow. Applying balancing to all WIs simultaneously reduces this growth dramatically.

5 – Swap by gradient

Map WIs with fewest updates to PEs with owned WIs that have the highest update gradient (number of updates completed in one time step). With PE imbalance only, this method works even better than the previous, see Figure 3.5. The spread does not exceed the expected minimum spread, i.e. the most that WIs can deviate in a single time step (Equation 3.1).



(a) PE variation. **(b)** WI variation. **(c)** PE and WI variation.

Figure 3.4: Swap all min and max policy.

$$\text{min spread} = \frac{\text{PE cycles} \cdot \text{max PE speed difference} \cdot \text{time step}}{\text{WI comp cost}} = \frac{4e6 \cdot 0.1 \cdot 0.01}{95} \approx 42 \tag{3.1}$$

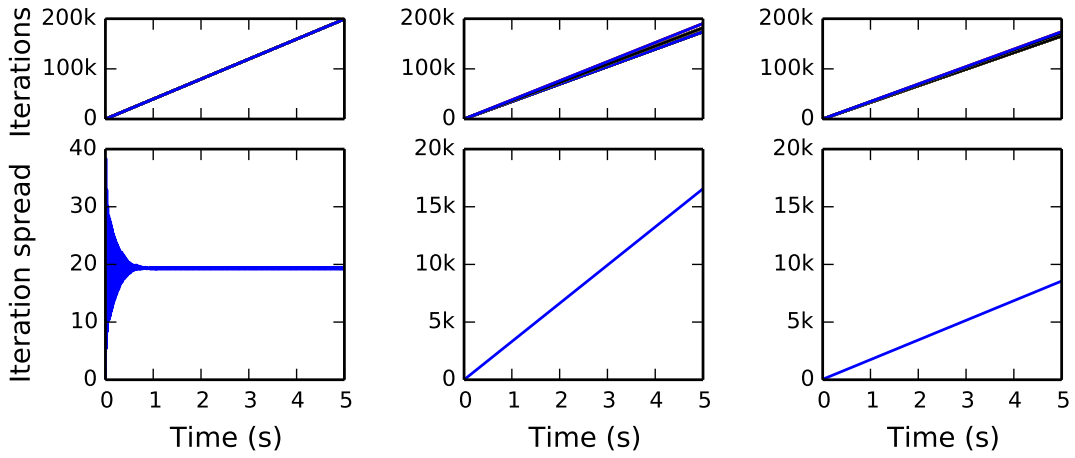
The exact limit (39) is slightly smaller in this case because, by chance, the difference between the fastest and slowest PE is closer to 0.09 than 0.1. However, the spread continues to oscillate and settle at about half that value. The limit assumes updates on two WIs starting at the same time and diverging as much as possible. Half of this limit would be achieved if one of the WIs was shifted by half of the spread. It appears that this policy slowly converges to such an "out of phase" scenario.

The policy does not help if there is only WI imbalance, but it does reduce spread significantly if there is both WI and PE imbalance. Differences in PE speed can be used to mitigate differences in WIs. Still, the spread grows continually, which is undesirable.

6 – Give min to max

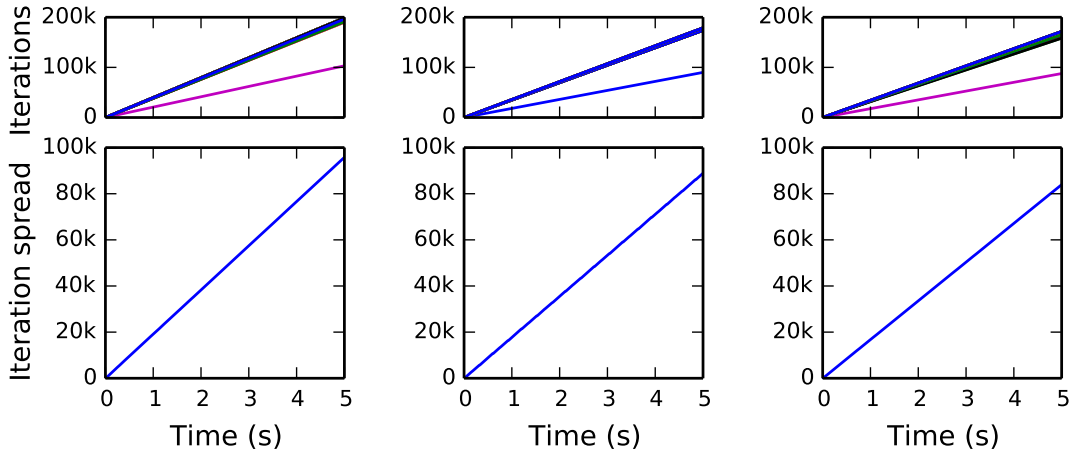
Put the least updated WI on the PE that holds the most updated WI, without receiving a WI in return. This policy was intended to start addressing the situation with WI load imbalance, but it is flawed and only increases spread, as can be seen in Figure 3.6.

After the transfer, the giving PE has no work and the taking PE has two WIs. Instead of improving the iteration rate of the WI that was given away, it is actually reduced because the WI is now on a more loaded PE. As a result, the same WI is the



(a) PE variation. **(b)** WI variation. **(c)** PE and WI variation.

Figure 3.5: Swap by gradient policy.



(a) PE variation. **(b)** WI variation. **(c)** PE and WI variation.

Figure 3.6: Give min to max policy.

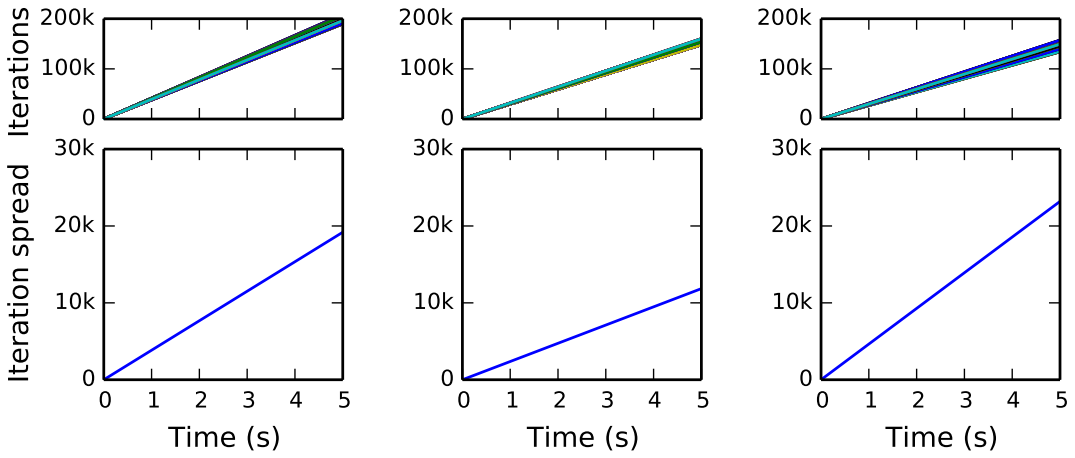
least updated in the next load balancing event and is given to another PE. With all PEs and WIs reasonably similar, the first WI to be given ends up with perpetually low iteration rate, hence the single diverging WI on the iteration plots in Figure 3.6.

While unsuccessful, this method did give the required insight for the subsequent policy.

7 – Give min to max with subsplitting

Subsplitting problem domains and carefully considering the effect of moving subdomains on iteration rates yields the main method presented in this thesis.

Each WI is split evenly along the X and Y directions, so each PE initially holds 4



(a) PE variation. **(b)** WI variation. **(c)** PE and WI variation.

Figure 3.7: No balancing. Each domain is split into 4 subdomains.

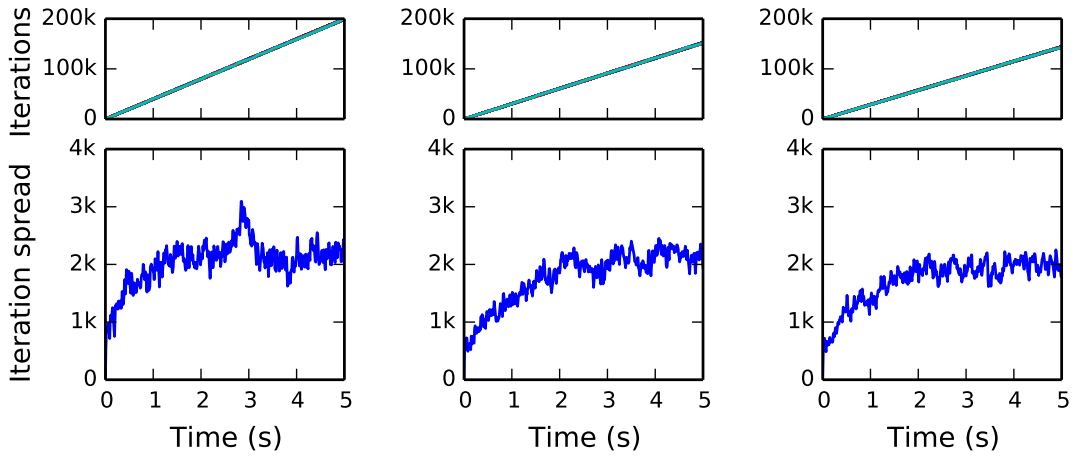
smaller WIs. The computational load of each WI is decreased linearly with its area; in this case – fourfold. The communication cost is also decreased linearly, i.e. by half for each neighbour. However, subsplitting domains increases the total amount of work spent on communication (there are more borders), so a new baseline is shown in Figure 3.7. Spread due to WI imbalance is reduced since the communication load difference between a corner and a bulk PE has reduced; corners have gone from exchanging 2 halos to 12 half sized halos, and the bulk has gone from exchanging 4 halos to 16 half sized halos.

When balancing, the most and least updated work items are found. Then a WI sharing the PE of the least updated WI is transferred to the PE holding the most updated WI. As a result, the least updated WI's iteration rate increases and the most updated one's iteration rate decreases.

This procedure can be applied to more than one pair of most and least updated WIs. Also, a threshold can be set for the minimum number of WIs on a PE. This prevents large "momentum" in iteration rates which tends to create overshoots by the time the next balancing period arrives and thus a larger spread. For the present simulations we compared 5 pairs of most and least updated WIs and each PE is limited to hold between 2 and 6 WIs.

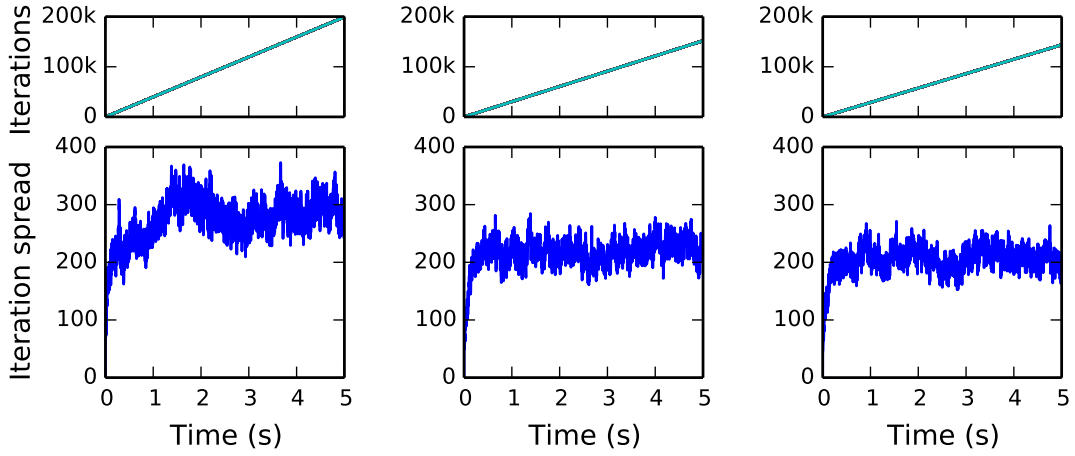
The results can be seen in Figure 3.8. This policy shows the ability to reduce and bound spread with workload imbalance and hardware performance variation present, both individually or together.

The spread can be easily bound further by increasing the load balancing frequency,



(a) PE variation. **(b)** WI variation. **(c)** PE and WI variation.

Figure 3.8: Give min to max policy. Each domain is split into 4 subdomains.



(a) PE variation. **(b)** WI variation. **(c)** PE and WI variation.

Figure 3.9: Give min to max policy with increased load balancing frequency. Each domain split into 4 subdomains.

e.g. from 100 times per second to 1000 times per second as shown in Figure 3.9. Doing so reduces the time frame in which the number of updates can diverge between WIs.

We performed further experiments where subsplitting was applied to the other balancing policies. Many of them can take advantage of subsplitting to some degree, but we found that method 7 takes advantage of it most reliably and effectively, since it is specifically designed for subsplitting.

We also repeated the experiments with different random initialisations of the simulated system. Method 7 performed consistently well, while others occasionally were significantly less effective at reducing spread due to an unlucky balance between WI

and PE noise.

Table 3.2 shows a summary and comparison of the data presented here. It adds a field for the number of WI moves that were performed in each case. In the simulator this does not matter, since we assume they are free, but in principle it is still desirable to minimise WI moves. Here method 7 also does well, so the WI moves are efficient in terms of achieving load balance.

Load balancing policy	Noise	# WI moves	Max spread	Mean spread
1 – No balancing	PE	0	19,531	9,765
	WI	0	16,861	8,430
	PE and WI	0	30,369	15,184
2 – Random balancing	PE	18,324	1,453	755
	WI	18,324	16,861	8,430
	PE and WI	18,324	16,823	8,529
3 – Swap min and max	PE	1,018	949	695
	WI	1,018	16,861	8,430
	PE and WI	1,018	15,670	7,791
4 – Swap all min and max	PE	18,324	81	47
	WI	18,324	16,861	8,430
	PE and WI	18,324	15,868	7,924
5 – Swap by gradient	PE	18,324	38	19
	WI	18,324	16,861	8,430
	PE and WI	18,324	8,709	4,373
6 – Give min to max	PE	1,018	97,492	48,721
	WI	1,018	90,535	45,194
	PE and WI	1,018	85,435	42,705
No balancing (subsplit)	PE	0	19,531	9,765
	WI	0	12,065	6,032
	PE and WI	0	23,612	11,806
7 – Give min to max (subsplit)	PE	2,119	3,093	2,045
	WI	2,130	2,451	1,794
	PE and WI	2,156	2,252	1,733
Give min to max (subsplit and more frequent balancing)	PE	21,141	372	276
	WI	20,965	284	216
	PE and WI	21,153	271	204

Table 3.2: Comparison of load balancing methods. The metrics of comparison are the mean and max of spread, as well as the total number of WIs that were moved during the run as a result of balancing. Each method is shown under 3 different conditions: noise on PEs, WIs or both.

3.2 Load balancing details beyond simulation

We were able to devise increasingly better load balancing policies for the case of PE noise only, but the added effect of WI noise was largely unrestricted. Subsplitting WIs gives more scope to address WI load variability. We use policy 7 – "give min to max with subsplitting" as the main balancing method due to its general effectiveness and name it "Progressive Load Balancing" (PLB). In this Section we expand on the practical details of the implementation of this method.

Our load balancing approach requires that the problem to solve can be split into more parts (i.e. WIs) than there are processing cores. For example, if a 2D iterative stencil application splits the problem domain equally among N CPU cores, we require that each domain is subsplit further on each core. This requirement is not imposing anything new on the application, as it would already have the requirement of domain decomposition in order to parallelise.

Each subdomain has an associated counter to keep track of how many times it has been updated. This information is used by the load balancer to decide which subdomain update rates need to be sped up and which need to be slowed down.

Subdomains start with some initial assignment to PEs (e.g threads) and are updated as normal. Threads are pinned to 1 core each. At set time intervals a load balancing function is run by one of the threads. This function decides how to reassign subdomains to threads based on differences in the number of updates to subdomains. Due to the coarseness of managing in units of subdomains rather than individual updateable elements, load balancing here is unlikely to result in a stable work distribution where further load balancing is not required. Instead it is continually adjusted so that the progress of subdomains is balanced when averaged over time.

Note that the balancing can be performed by any thread on a node. Thus the impact of balancing decision calculations can be spread evenly in the machine. Also, the balancing can itself happen asynchronously by querying the progress of different WIs while they are being updated. This offers significant scalability advantages over centralised approaches, such as placing all WIs in a priority queue where priority is assigned based on a WI's staleness.

3.2.1 Implementation

PLB can be implemented as is detailed in the listing Algorithm 5. In principle, the algorithm works on both shared and distributed memory, however here we present a

```
Data: nPairs, lowThresh, highThresh
1  doms ← list of subdomains sorted by update count (descending);
2  for i ← 0 to nPairs do
3      topSubd ← doms[i];
4      botSubd ← Reverse(doms)[i];
5      topThread ← topSubd.GetThread();
6      botThread ← botSubd.GetThread();
7      if topThread.GetNumSubds() < highThresh and
         botThread.GetNumSubds() > lowThresh then
8          subdToSend ← FindMostUpdatedSubd(botThread);
9          subdToSend.SetThread(topThread);
10     end
11 end
```

Algorithm 5: Progressive load balancing.

shared memory implementation as a proof of concept.

The intuition behind the algorithm is as follows: each thread updates a number of subdomains and it is possible to increase or decrease the update (or iteration) rates of those subdomains by removing subdomains from a thread or assigning more to it, respectively. However, reassignment of subdomains to different threads must be done carefully and requires considering the effect this will have on the progress of other subdomains belonging to the affected threads.

In more detail: if a subdomain `topSubd` has had the most updates, the thread that it belongs to, `topThread` is likely fast (e.g. because it is pinned to a core not experiencing any noise or it has less work). The load balancer will reduce the iteration rate of `topSubd` by slowing down thread `topThread` through giving it an additional subdomain to work on. At the same time, we wish to increase the iteration rate of the subdomain `botSubd` that has had the least updates. We find the associated thread `botThread`, which owns `botSubd`, and pick a subdomain from it *other than* `botSubd` and reassign this to `topThread`. Now that `botThread` has one fewer subdomains to update, the iteration rate of `botSubd` will increase.

The iteration rate of the subdomain that is reassigned may go up or down, depending on the relative number of subdomains on `topThread` and `botThread`. The algorithm therefore picks the subdomain that has had the most updates on `botThread` to move to `topThread`, because that subdomain will usually be close to the average in

terms of updates completed, so it is unlikely to race too far ahead or fall behind. This choice meets the requirement that the transferred subdomain is not equal to `botSubd`, if there are at least 2 subdomains on `botThread`. Repeatedly performing this load balancing achieves a progressive oscillation of iteration gradients, hence limiting spread of updates per subdomain.

We implemented this algorithm in the C programming language and compared the output with the results obtained from simulations. Figure 3.10 shows a section of the iteration count plot in the middle of both a simulated and a real run. The same oscillation pattern can be seen in both cases, thus validating the simulation approach. We call these repeated oscillations **"braiding"** – the unique pattern of subdomain update progress that emerges when PLB is applied to an asynchronous algorithm. Braiding is key for limiting progress variability while maintaining the performance benefits of asynchrony.

A detailed evaluation of the load balancing method using this implementation is set out in Chapter 4.

3.2.2 PLB parameters

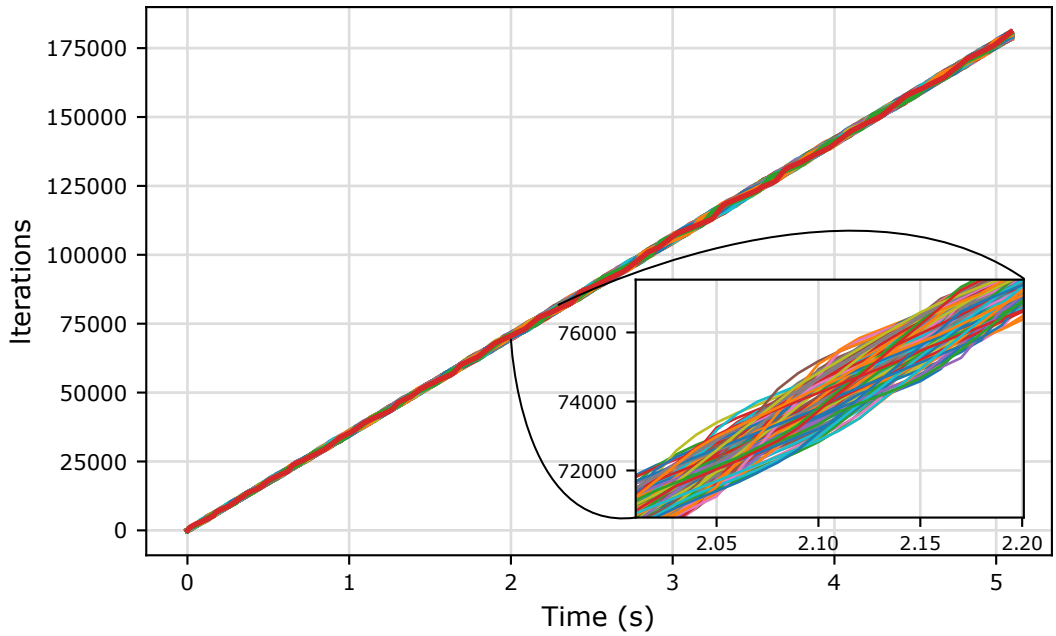
PLB has a number of parameters that impact how effective the balancing is.

Degree of subsplitting. In general, splitting domains into more subdomains gives greater ability to reduce spread, but it may reduce performance. For example, if the application does stencil computation, performance would decrease due to higher communication frequency between subdomains to exchange halos and less contiguous cache use.

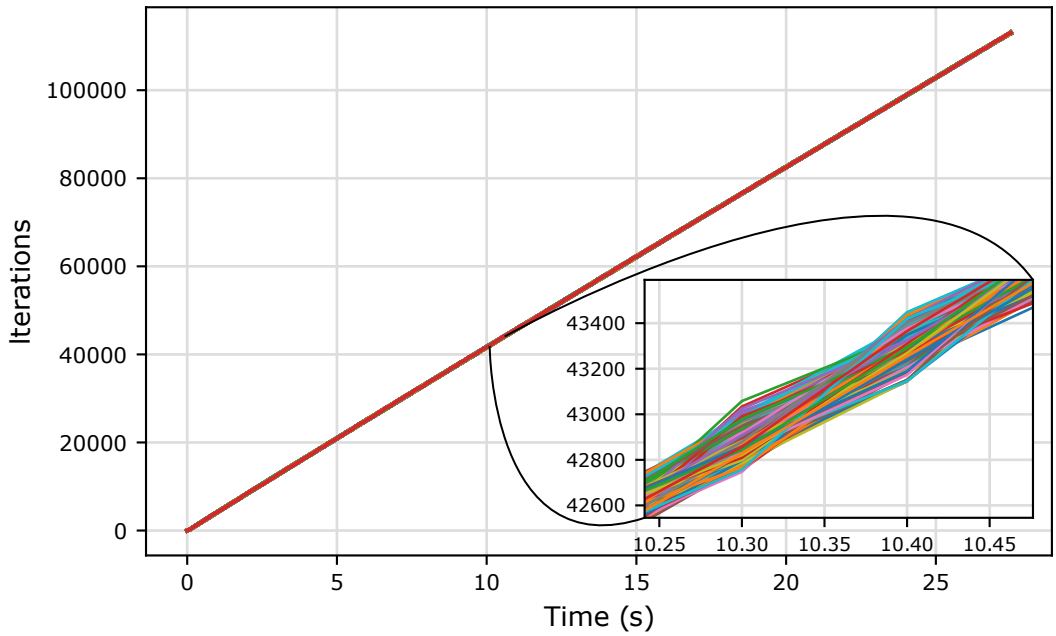
Load balancing frequency. The higher the frequency, the tighter the bound on spread. However, the cost incurred by moving WIs also increases.

Number of pairs. Considering more than one pair of subdomains for balancing (Line 2 in Algorithm 5) helps the algorithm load balance the whole problem, rather than one part of it. The pairs are chosen as the top nth and bottom nth subdomains, ordered by the number of updates completed. A higher number of pairs that get compared in each load balancing event decreases the spread. The more work items there are the more numbers of pairs one needs to compare.

Upper and lower thresholds. The two thresholds (Line 7 in Algorithm 5) are to provide some "inertia" and to avoid large swings in iteration rates. They set the highest and lowest number of WIs that a PE is allowed to have. If the thresholds allow



(a) Simulated



(b) Real

Figure 3.10: A comparison between simulated and real PLB. Problem subdomains (represented by one coloured line each) are being moved between slow and fast running threads to ensure an overall even progress towards the solution. The lines in the real case appear to change gradient less often than in the simulated case due to the rate of sampling, which was kept low to minimise overhead.

a wide range, the balancer can affect spread to a greater extent but also is in danger of overshooting. A high range requires a suitably large balancing frequency in order to be able to negate any overshoot.

3.2.3 Variants due to hardware

Since the cost of moving data within a CPU or across sockets is different, we can develop 3 variants of the algorithm:

Joint All cores are treated the same. Subdomains may be moved across sockets.

Split CPUs on different sockets are balanced separately.

Hybrid CPUs on different sockets are balanced separately. Periodically (this is a tuneable parameter) a random subdomain is moved from a thread on the socket that has completed the least updates on average to a thread on the socket that has done the most.

3.3 Summary

In this Chapter we described how the problem of load balancing an asynchronous algorithm can be modelled for simulation. We then used the simulator to develop a series of load balancing methods. It was found that problem domain subsplitting is required to achieve a versatile load balancing method that can handle both hardware and workload variability. The method – progressive load balancing – was then described in more detail including how to implement it. In the next Chapter we will use this implementation to evaluate PLB in a shared memory context.

Chapter 4

PLB in shared memory

In this Chapter we evaluate progressive load balancing (PLB) in a shared memory context with CPU core performance variability present. The aim is to maintain a staleness boundary dynamically, which is expected to lead to improved performance and insensitivity to noise.

Using Jacobi iterations as a test case, we show that update spread is bounded under a variety of scenarios. At the same time, the overhead of balancing is small in most cases, so the high iteration rate enabled by asynchrony is maintained. As a result of lower progress imbalance and higher iteration rate, the balanced asynchronous method outperforms synchronous, semi-synchronous and totally asynchronous implementations in terms of time to solution. Hence, the impact of noise is minimised. In contrast, work stealing is shown to be ineffective.

4.1 Methodology and experiments

In this Section we describe our approach to comparing asynchronous algorithms with different synchronisation types as well as the specific setup of our environment and experiments used to evaluate the progressive load balancing approach.

4.1.1 PLB and Jacobi implementation

The load balancer and the Jacobi application are implemented using the C programming language and OpenMP [78] for general purpose threading and synchronisation primitives. The application calculates grid updates as described in Section 2.3.2; calculations use double precision floats. Two arrays are used for each subdomain to hold

the current and previous state and the copy between iterations is done explicitly. All subdomains are shared so that any thread can work on any subdomain; this facilitates load balancing via PLB. Each array that holds a subdomain is padded with an extra layer of elements around the perimeter; halos from neighbouring domains are copied into the padding when halos are retrieved. This explicit copying allows all elements of the subdomain (i.e. the inner elements of the array) to be updated in the same way without requiring special treatment for edge elements. Halos are retrieved in a "racy" manner as introduced in [65], i.e. the halos are read directly from the boundaries of neighbouring problem domains, thus minimising synchronisation. If the run type is semi-synchronous, halo staleness is checked and the iteration skipped if one or more halos are too stale. The residual is periodically updated for each problem domain and asynchronously summed to find the global residual to test if the stopping criterion is met.

The PLB algorithm is implemented according to Algorithm 5. Threads take turns (round robin) to call the load balancing routine at a set frequency and proceed to re-assign subdomains to threads as decided. Threads that are affected by the balancing (either gaining or losing work) are marked as dirty. At the start of an iteration, each thread checks whether its working set has been dirtied before computing updates. If it has, the affected thread takes note of the new working set of subdomains and proceeds with calculating updates.

4.1.2 Evaluation metrics

Our general approach is to compare an asynchronous algorithm with progressive load balancing against synchronous, semi-synchronous and asynchronous implementations of the same algorithm. Additionally, in Section 4.2.4 we provide a comparison with work stealing – a popular load balancing method. We evaluate the different implementations using three metrics:

Iteration rate The number of iterations completed per second. In an asynchronous setting this value is not straightforward, so we define it as the total number of iterations across all threads divided by the number of threads.

Spread A measure of the upper limit of progress imbalance. It is the difference between the maximum and minimum number of updates completed on subdomains of the problem at the end of the run.

Time to solution Time taken to reach a chosen level of accuracy of the solution.

We believe it is instructive to consider iteration rate and a measure of staleness separately (supplementing time to solution) when evaluating any asynchronous algorithm; time to solution is a function of the two metrics and reducing staleness also helps with stability.

4.1.3 Measurement and hardware setup

Asynchronous algorithms are stochastic in nature, so it is important to take many performance samples, including multiple different nodes, to get a full picture of the range of performance. Each different variant of the application run that we present here was repeated 100 times on 3 nodes making for a total of 300 samples per experiment. The data from the 3 nodes was aggregated. On production HPC systems it would have been extremely time consuming to get the same set of nodes across all the experiments we ran. Instead, we used randomly assigned nodes which resulted in shorter queuing times as well as giving a more representative landscape of performance.

The experiments were run on 2 production HPC systems of different generations, with details shown in Table 4.1.

4.1.4 Test problem and load balancing settings

We chose Jacobi's algorithm in 2D as the test application for our evaluations; see Section 2.3.2 for details about the problem. This algorithm meets asynchronous execution stability requirements [7] which means that we did not have to worry about failure to converge. It is therefore suitable as a comparison point across different synchronisation types, which made it possible to focus on the performance and load balancing aspects of the investigation. Additionally, its simplicity facilitates quick implementation and analysis of different variants. This allows us to perform an in-depth analysis of PLB (Chapters 4 and 5); further application areas will be discussed in Chapter 6. While Jacobi's algorithm is not in wide production use, we believe that our findings are transferable to other stencil applications or different asynchronous algorithms (e.g. ones listed in Section 2.3.3), because PLB has been designed to be a generic method and has not been specialised for Jacobi's algorithm.

We used Dirichlet boundary conditions as in [65]. The boundary conditions used were all zeros on 3 sides of the global domain and a Gaussian shaped source on the

Table 4.1: Machine and compilation details.

	ARCHER	Cirrus
System type	Cray XC30	SGI ICE XA
CPU Sockets	2	2
CPU	Intel E5-2697 v2	Intel E5-2695
Core count per CPU	12	18
Clock	2.7 GHz	2.1 GHz
Architecture	Ivy Bridge	Broadwell
Interconnect	Cray Aries	FDR Infiniband
Topology	Dragonfly	Hypercube
Nodes	4920	282
L3 cache	30 MB	45 MB
RAM per CPU	32 GB	128 GB
Compiler	CCE 8.5.2	GCC 6.2.0
MPI library	Cray MPICH 7.2.6	Intel MPI 16.0.3
Main compilation flags	Cray default (-O2)	-O2
Noise gen. flags	-O0	-O0

4th side defined by $e^{-s(0.5-x)^2}$ where s is a scaling constant and x is the horizontal coordinate in the global grid. The domains were sized 300×300 cells per thread and initialised to a constant value of 1 in each cell. On Cirrus the cores were arranged on a 6×6 grid making the global problem size 1800×1800; on ARCHER the arrangement was 6×4, which makes the global problem size 1800×1200.

The termination criterion for iteration rate and spread experiments was one thread reaching 5000 iterations, or 5000 multiplied by the number of subdomains each domain was split into. For time to solution experiments the criterion was reaching 10^{-4} global l_2-norm of the residual normalised by the initial global l_2-norm of the residual, defined as $r = b - Ax$.

The additional load balancing parameters described in Section 3.2.2 were set as follows. Each domain was subsplit into 4 subdomains (by halving in both the x and the y directions); we found this to give a good balance between balancing power and performance overhead. The load balancing lower threshold was set to 2, the upper threshold was set to 6 (a modest offset from the base number of subdomains per thread to avoid large iteration rate swings) and 6 subdomain pairs were considered for moving. The frequency of balancing was set to 100 or 1000 times per second and, in the hybrid case, cross socket balancing was set to occur after every 50 or 500 balancing events (the parameters are specified for each experiment). These parameters were empirically found to give reasonable performance in the majority of cases. Other settings were explored both in the simulator and pilot experiments, but they were generally found to be less performant or less versatile.

4.1.5 Simulating noise

Performance variability is required to test the load balancing schemes. There is some workload inequality due to varying numbers of halo exchanges between domains. In addition to this we inject noise in a consistent manner across experiments to be able to discern the effects of different synchronisation types. In these experiments we simulated only one noisy core. This is a minimum case and illustrates the weakness of bulk synchronicity as the whole node is affected by a single slow thread.

To generate noise we are running a parasite process in the background (due to job scheduling specifics, on ARCHER this is a process and on Cirrus a thread within the main application). The background application (pinned to one core) switches between sleeping for 200 microseconds and performing 10000 iterations of $sum = sum \cdot a + b$.

On both machines the work loop, when run in isolation, takes on average 46 microseconds to compute so the noisy core is about 19% slower than the rest ($\frac{46}{46+200} \approx 0.19$). This level of CPU frequency variation can be reasonably expected when applying even a small power cap [28]. For the experiments presented in this Chapter, the noise generator has the same priority as the main application, so OS scheduling may have an effect on the exact level of noise seen in practice.

In addition to the amount of noise, noise placement has an effect on the time to solution. Threads are usually pinned to cores in HPC applications because doing so removes time wasted due to thread migration. Given that the problem is decomposed based on threads, a noisy core would affect a particular part of the problem domain. Which domains are sensitive to noise is problem dependent, but, given a complex set of equations, it could be most of the problem space. In our experiments, to demonstrate the effect on convergence time, we placed the noise next to the boundary that contains the source, as this domain was found to be most sensitive to stale values.

4.2 Evaluation

In this Section we present experiment results and evaluate and compare the different synchronisation types based on the metrics defined in Section 4.1.2.

Figures 4.1 and 4.2 shows how iteration rate and spread compares across synchronisation types. The best methods are in the lower right corner, i.e. the aim is to minimise spread *and* maximise iteration rate. Also, the the best methods will not change position on the plot by much when noise is added.

Figures 4.3 and 4.4 show the time to solution of a representative subset of methods. Less time and smaller variance is better.

The different synchronisation types are denoted using the following labels:

sync synchronised by global barrier

ssync(i) halos from neighbouring domains must be within i iterations of the updating cell

async(n) totally asynchronous version with each domain subsplit into n subdomains

Load balancing types are specified by:

s(f) each CPU socket is balanced independently every f seconds (s stands for split)

j(f) all CPU sockets are balanced together every f seconds (j stands for joint)

h(f, n) each CPU socket is balanced independently every f seconds and cross socket adjustments are done every nth balancing (h stands for hybrid)

4.2.1 Spread reduction

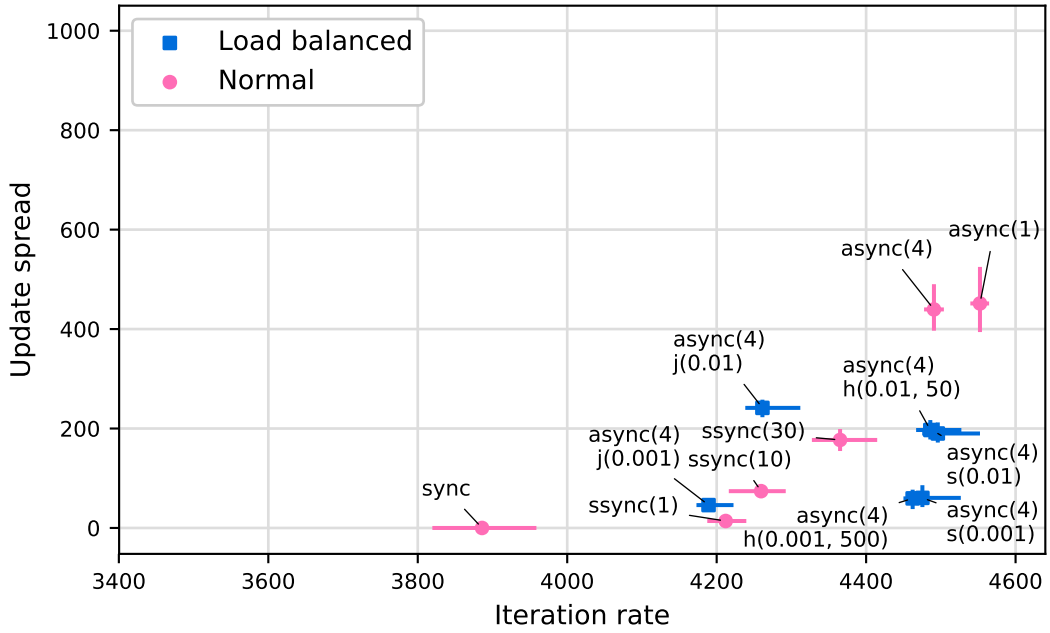
Adding load balancing to asynchronous Jacobi decreases update spread, with higher load balancing frequency resulting in lower spread in all cases (see Figures 4.1 and 4.2). Invoking the balancer more often results in a tighter "braid", as illustrated in Figure 3.10, thus reducing spread.

Note that update spread for the semi-synchronous variants is higher than their stated staleness limit. This is to be expected, because the limit is between neighbours, but spread is measured globally.

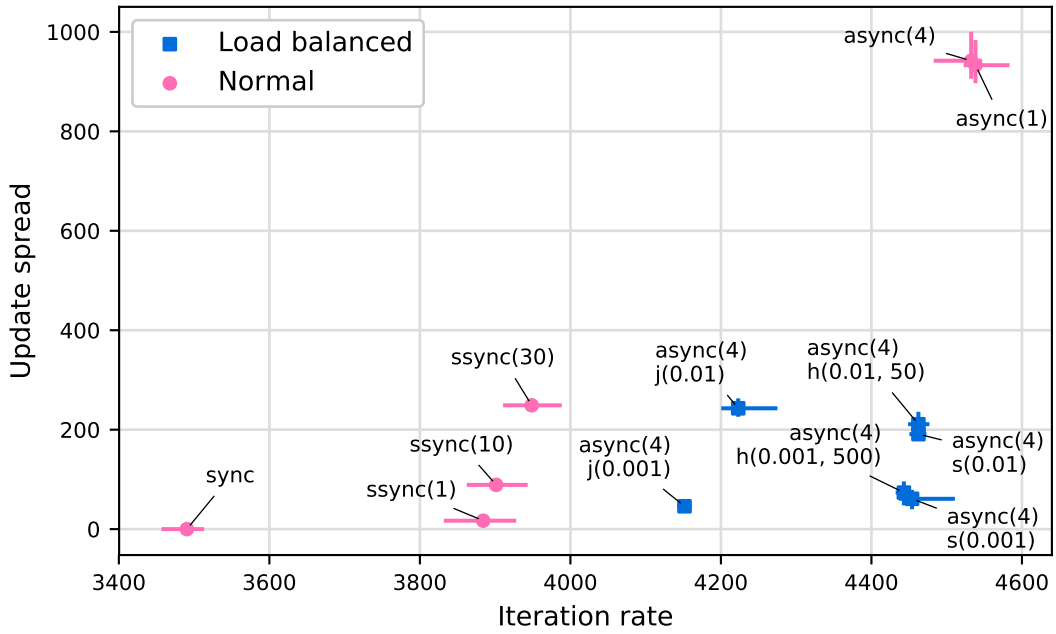
The spread of load balanced versions is comparable to, or lower than, the semi-synchronous versions, except for ssync(1). Among the load balanced versions, the joint scheme achieves the lowest spread. The split and hybrid schemes follow closely, however it should be noted that the split scheme would slowly grow in update spread with increased iteration count while the joint and hybrid schemes would remain steady. This is the case because the split scheme does not exchange subdomains between CPU sockets, so load balancing is done with respect to each socket separately.

Adding noise to a core has the least impact on spread when using a load balanced scheme. Across the two machines, this ranges from a *decrease* in spread of 38% (ARCHER, async(4) + hybrid(0.001,500)) to an increase by 24% (Cirrus, async(4) + hybrid(0.001, 500)). The semi-synchronous schemes increased by between 8% (ARCHER, ssync(1)) and 76% (ARCHER, ssync(30)), while the totally asynchronous schemes range between an increase of 107% (Cirrus, async(1)) to 443% (ARCHER, async(1)).

It is an interesting observation that some of the balancing variants performed better with added noise. We postulate that these variants benefit from more gradual iteration rate changes. For example, removing a subdomain from the noisy thread would make the iteration rates increase by a smaller amount than if the subdomains were on a normal thread, thus avoiding overshooting the spread bound.

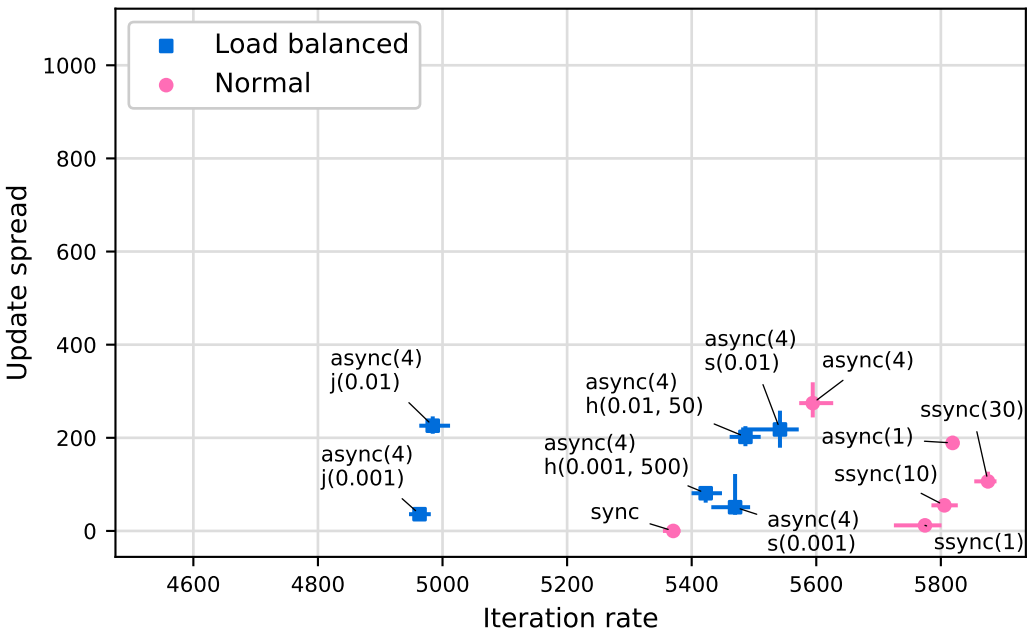


(a) No added noise

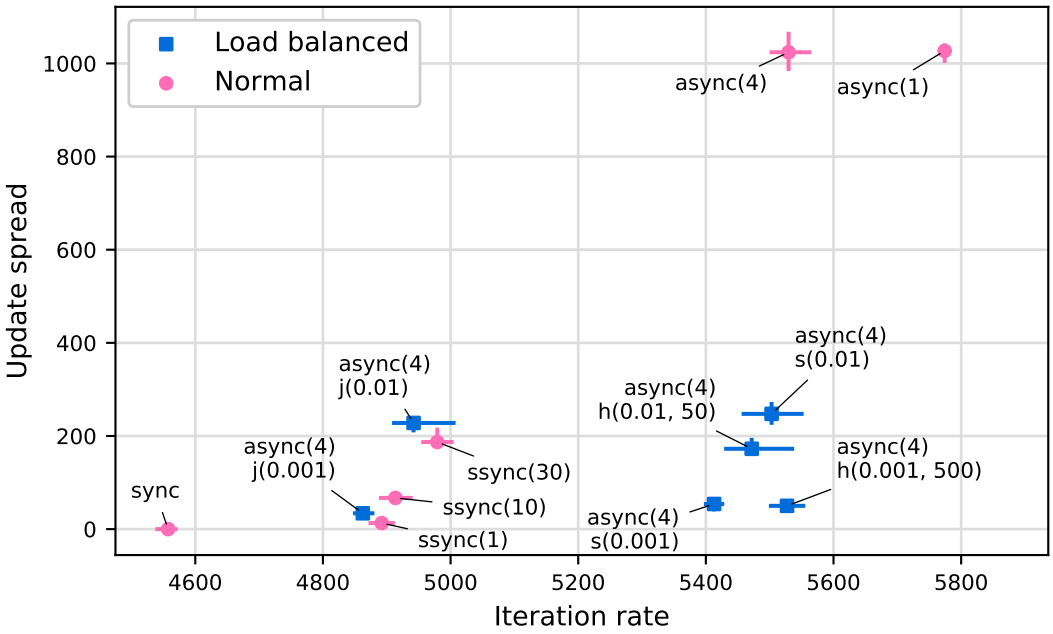


(b) With added noise

Figure 4.1: Landscape of synchronisation types on Cirrus. The best synchronisation methods are in the bottom right corners (high iteration rate and low spread), and do not change significantly when noise is added. The points represent median values and the error bars show the 25th and 75th percentiles.



(a) No added noise



(b) With added noise

Figure 4.2: Landscape of synchronisation types on ARCHER. The best synchronisation methods are in the bottom right corners (high iteration rate and low spread), and do not change significantly when noise is added. The points represent median values and the error bars show the 25th and 75th percentiles.

4.2.2 Iteration rate

The best performance in terms of iteration rate is achieved by totally asynchronous methods, though ssync(30) on ARCHER is an exception (Fig. 4.2a).

The overhead of load balancing varies significantly by method. We define the overhead as the percentage difference between the median iteration rate of async(1) and a load balanced version; the reported range includes all parameter variations of the specified version and experiments both with and without added noise. On Cirrus, split balancing has an overhead of 1%–2%, joint 7%–9% and hybrid 1%–2%. On ARCHER, split balancing has an overhead of 5%–7%, joint 17%–19% and hybrid 4%–7%; it should be noted that the subsplitting itself introduces a 4% overhead on ARCHER and up to 1.5% on Cirrus (async(4) compared to async(1)).

The joint policy has the worst iteration rate due to data movement across CPU sockets. The hybrid policy mitigates this issue, often providing performance similar to or exceeding the split policy. In all cases increasing load balancing frequency has a negative impact on iteration rate.

Importantly, the load balanced versions are not heavily affected (less than 1% slowdown on Cirrus, and up to 2% on ARCHER) by the addition of noise – it is effectively spread out among the cores. The addition of progressive load balancing retains the essential property of asynchronous methods to resist noise. Synchronous and semi-synchronous methods are very sensitive to noise and largely become slower (8%–10% on Cirrus, 15% on ARCHER) than load balanced asynchronous variants.

Overall, the relative performance of different synchronisation types is affected by both algorithm settings and machine parameters, as evidenced by the different positions of equal points in the landscape plots (Figs. 4.1 and 4.2). Nevertheless, the presented analysis points out trends that are generalisable and load balancer overhead is low in most cases.

4.2.3 Time to solution improvement

Figures 4.3 and 4.4 give an example of the benefits of the progressively load balanced asynchronous approach. When the systems are running normally, the asynchronous methods converge in the least time. The version with split load balancing follows closely behind. If noise is added to the systems, there is a drastic difference in time to solution response, as summarised in Table 4.2. The synchronous and semi-synchronous versions take the longest to converge; they are limited by the slow-

Table 4.2: Time to solution increase of different synchronisation types when adding 19% noise to one core. Cirrus has 36 cores on a node and ARCHER has 24.

	Cirrus	ARCHER
Normal		
sync	13%	18%
ssync(1)	11%	18%
ssync(30)	11%	18%
async(1)	8%	10%
async(4)	9%	11%
Load balanced		
async(4) + s(0.001)	1%	<1%
async(4) + j(0.001)	1%	1%
async(4) + h(0.001, 500)	-2%	1%

est component. It is worth noting that the slowdown on Cirrus is smaller than on ARCHER; this is likely due to differences in OS level scheduling of the noise generating thread (on Cirrus) and process (on ARCHER). The totally asynchronous versions leave behind the slow running thread thus maintaining their iteration rate, but these iterations are less useful due to the increase in update spread. Adding load balancing successfully mitigates the negative effect of the noisy core. The balanced versions effectively distribute the penalty of one slow core across all available cores on the node.

As a result, with hardware performance variability present, our best load balanced method resulted in 22%–25% speedup over synchronous, 14%–19% speedup over semi-synchronous and 5%–8% speedup over totally asynchronous schemes.

Based on the landscapes in Figures 4.1 and 4.2, we expected the split(0.001) load balanced version to converge the quickest on Cirrus and hybrid(0.001, 500) on ARCHER. The results on Cirrus (Fig. 4.3) met our expectation, but on ARCHER (Fig. 4.4) they did not; instead, the split(0.001) balancer gave the quickest convergence again. It appears that update spread, while useful and simple to evaluate, has some limitations when used as a measure of optimum load balancing.

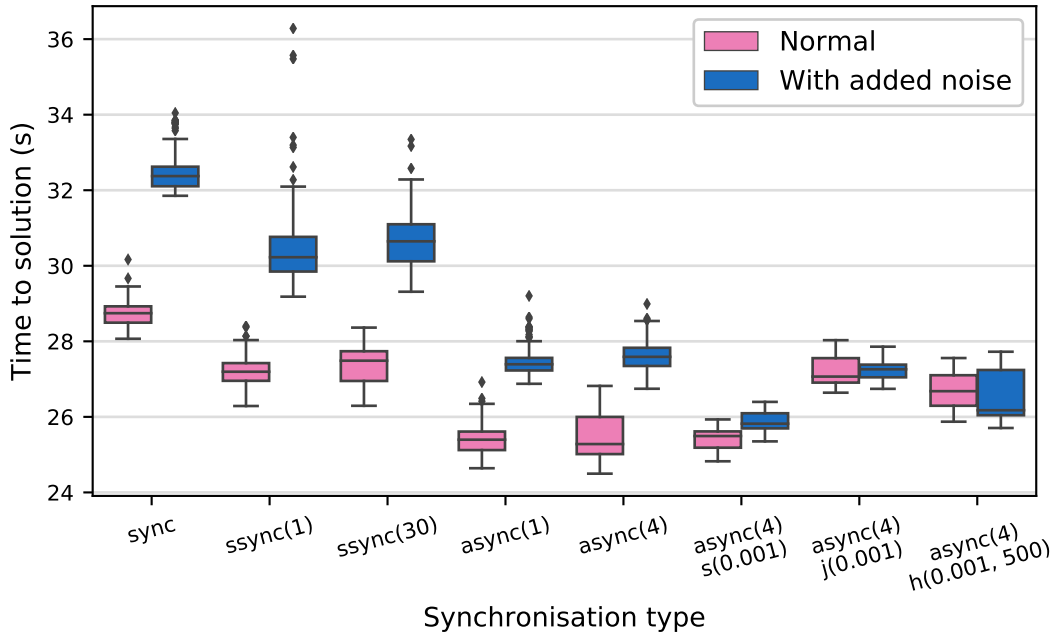


Figure 4.3: Time to solution on Cirrus. The load balanced versions are least sensitive to noise.

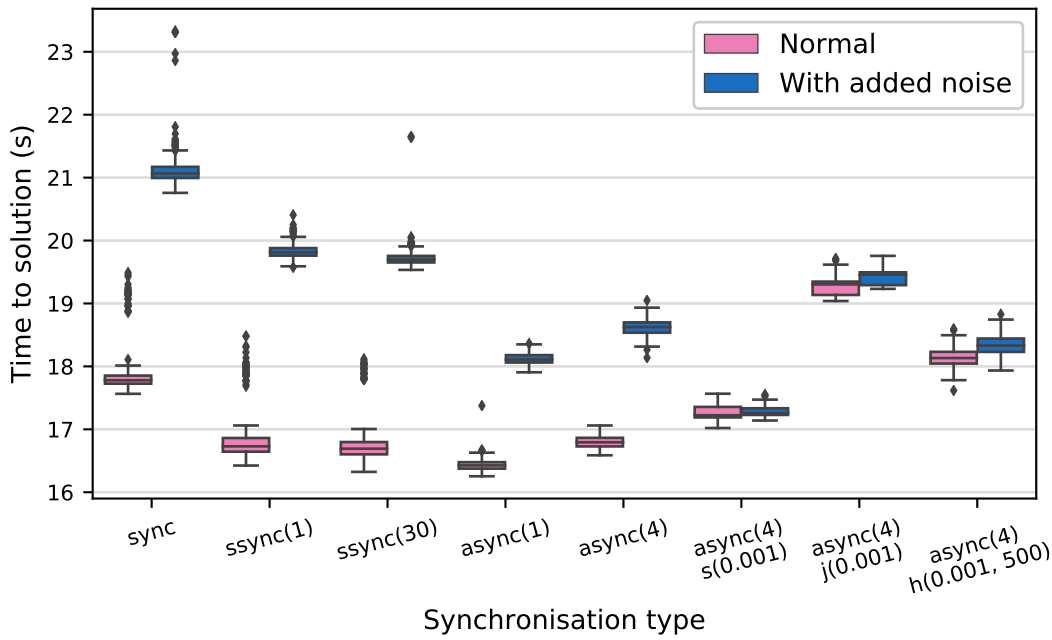


Figure 4.4: Time to solution on ARCHER. The load balanced versions are least sensitive to noise.

4.2.4 Comparison with work stealing

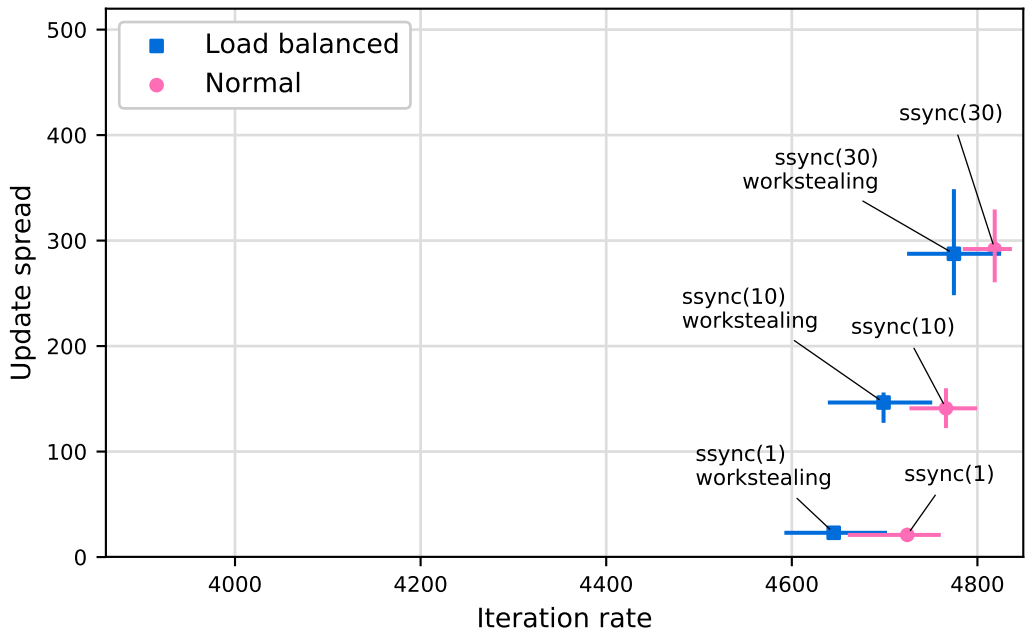
As discussed in Section 2.3.4, existing load balancing algorithms are generally not designed to be applicable to asynchronous algorithms. However, we adapted work stealing [79] to provide a point of comparison. In this method each worker completes their own list of work first before choosing a random other worker and stealing an item from their list of work.

The implementation is not obvious because each worker in an asynchronous algorithm always has "available work" because it can continue iterating using the latest available data, even if it is stale, and thus would not have a need to steal work. However, work stealing can be applied to the semi-synchronous variant if one interprets reaching a point where all halos are too stale to do an update as the work list being empty. Therefore, we evaluated work stealing applied to semi-synchronous Jacobi.

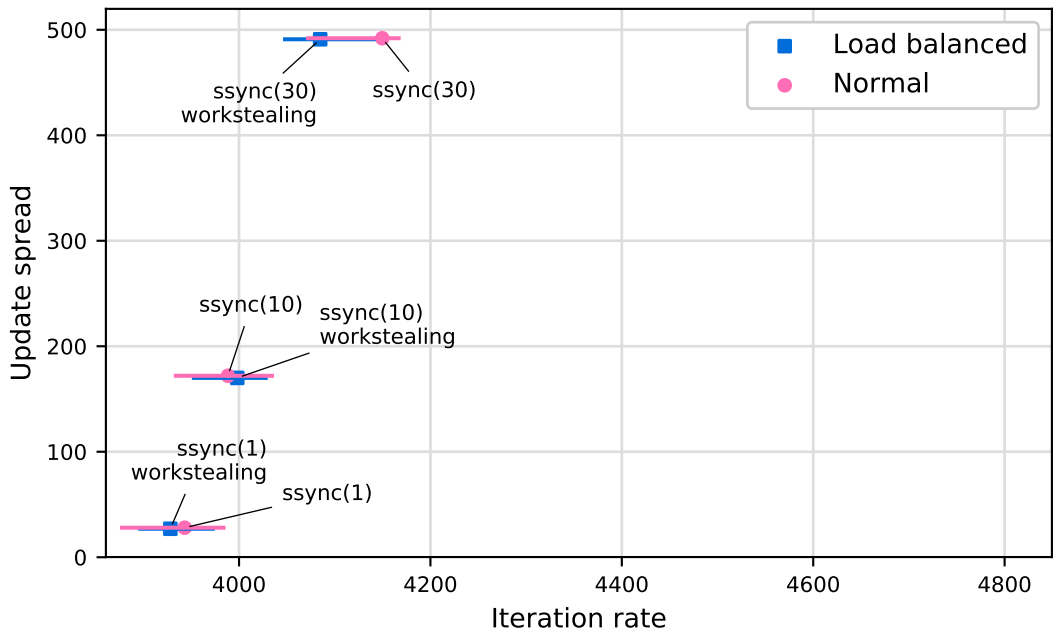
We ran our tests on Cirrus using the same problem setup and base code as before. Each unique experiment was repeated 10 times on 3 different nodes, giving a total of 30 samples. The problem domains are subsplit into 4 subdomains in order to allow work stealing to take partial load off lagging threads. Threads prioritise the set of subdomains assigned to them first; if all their work items' halos exceed the staleness limit, attempts are made to update randomly chosen subdomains which belong to other threads. A thread checks whether it has some new work of its own after it completes a successful steal or fails to steal 100 times in a row.

Figure 4.5 shows update spread and iteration rate for three different staleness boundaries. It can be seen that performance with added work stealing is similar to the corresponding versions without. Update spread is almost the same but iteration rate is usually lower with work stealing.

In terms of absolute numbers, the iteration rates in this set of experiments are about 10% higher than in the experiments presented in Figure 4.1. As a control, we tested the asynchronous variant and a similar rise in performance was noted. The likely cause of this difference is a change in the available compiler suite (Intel version 18) on Cirrus due to performing these experiments at a later date. Since the change appears to be universal and since we are primarily interested in relative change, i.e. how work stealing changes performance, we do not show these values side by side with the previously examined Jacobi variants. We also considered that subsplitting may affect performance due to changing the size of the working set within each work item, but measurements showed no significant difference between having and not having subsplitting.



(a) No added noise



(b) With added noise

Figure 4.5: Performance metrics of work stealing on Cirrus. The best synchronisation methods have high iteration rate and low spread. The points represent median values and the error bars show the 25th and 75th percentiles. Work stealing has a negative effect on performance and does not reduce update spread.

Absolute update spread is also higher in the present experiments, but that is explained by the addition of subsplitting. The maximum spread is determined by the number of neighbours between the slowest worker and the furthest normal worker. By splitting domains in half in the x and y directions, the distance has been doubled and hence the update spread has doubled as well. Spread is lower in the case without added noise because staleness buildup is slower with a smaller iteration rate gap between workers, so the maximum spread may not be reached (run time is limited).

Figure 4.6 shows work stealing results in terms of time to solution. The versions with work stealing took slightly longer to converge on average, reflecting the added overhead without signifiant effect on staleness. The results with added noise show that sensitivity to noise was not decreased by work stealing.

As previously stated in Section 2.3.4, once the staleness boundary has been reached in most of the system, there is little work to be stolen and there are many starved workers. On average, without added noise 0.2% of updates completed by a thread were stolen work and with added noise – 0.6%. Therefore, the variant with work stealing does not differ much from plain semi-synchrony in practice. Each time a work item is executed on the stalling worker, a "wave" of new work is released from neighbour to neighbour. However, since workers prioritise their own tasks first, the new work is quickly consumed and the previous situation resumes. An improvement would be for starved workers to steal from the straggler for a period of time, thus releasing much more new work into the system. However, this would require significant departure from what is classically called work stealing and would in fact begin to resemble PLB. Therefore we do not consider work stealing as a point of comparison any further.

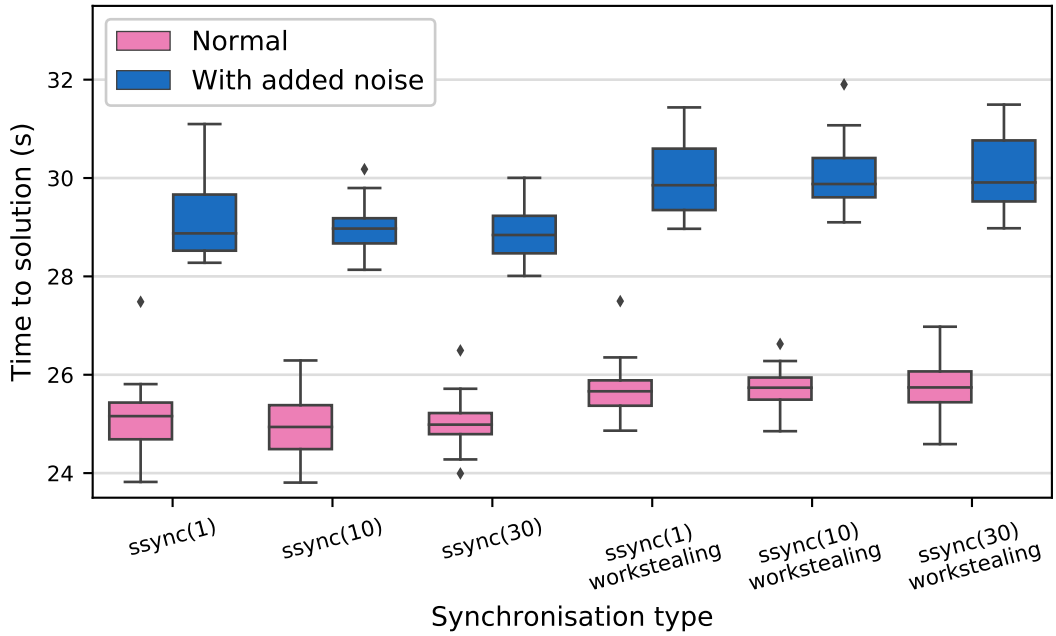


Figure 4.6: Time to solution on Cirrus. Work stealing does not improve performance or mitigate noise.

4.3 Summary

We have evaluated progressive load balancing in shared memory on two HPC systems. Using Jacobi's algorithm as a test case, we have shown that an implementation of this method lowers update spread while maintaining a high iteration rate under most settings, especially in the presence of noise. As a result, our load balanced method achieved a 5%–25% speedup in terms of time to solution over other synchronisation types, with 19% noise added to one core. In contrast, work stealing was shown to be ineffective.

More performance variability is expected in the distributed memory case where data transfer has to take place across the network and where whole nodes can be affected by noise. We investigate the application of PLB in this setting in the next Chapter.

Chapter 5

PLB in distributed memory

We have shown that PLB is able to effectively mitigate the effect of a slow core in a shared memory environment. PLB achieved this by periodically moving work between CPU cores, not in order to equalise iteration rates, but to bound progress imbalance; load is balanced over time, not instantaneously. In this Chapter we build upon PLB and present and evaluate an extension to the distributed memory setting.

We first test an approach where we run independent load balancing on each node and show that this reduces progress variability in cases where system noise is symmetric across nodes. For the more general case, we describe and evaluate a strategy where load balancing is allowed to also take place between nodes. We demonstrate that this method is able to mitigate system performance variation by reducing global progress imbalance, time to solution and time to solution variability.

5.1 Extending PLB to distributed memory

While PLB was shown to be successful in a shared memory setting, for it to be truly valuable it needs to be able to scale further. In this Section we present two ways to apply PLB in the distributed memory setting.

5.1.1 IPLB

The straightforward extension to support distributed memory is to run PLB on each node separately, while the main application is solving a problem distributed across nodes. We refer to this method as *independent progressive load balancing* (IPLB). No work is moved between nodes, only between cores on each node.

Applying independent instances of PLB reduces progress variability on nodes by effectively averaging noise across available cores. If performance variability is mainly caused by noisy cores or OS tasks assigned to some cores, IPLB can smooth it out and, as a result, move towards global progress spread reduction.

5.1.2 DPLB

As a further extension we also consider moving work between nodes (Fig. 5.1). This method is referred to as *distributed progressive load balancing* (DPLB). In DPLB we still run PLB on each node, but add infrequent work movements across nodes. This extension is important for situations where whole nodes are affected by noise and are significantly slower than others.

The main steps in the algorithm are as follows:

1. Periodically, with a set frequency, nodes check the average number of updates performed on other nodes.

2. The difference between the highest and lowest averages are compared to a set threshold.

3. If the difference is larger than the threshold, the least progressed node sends a randomly chosen problem subdomain to the node that has advanced the most.

4. The node that has received the subdomain assigns it to one of its cores initially, but, since PLB is running on every node, the subdomain gets moved between cores as is required to balance progress on the node.

The implementation details of these steps will vary based on the problem that is being solved, and the programming techniques and libraries used. The next Section explains some of the most important implementation considerations for our example case.

5.1.3 Implementation

Distributed communications are mainly implemented using MPI single sided calls. This communication paradigm is well suited to asynchronous algorithms, since it minimises the need for global synchronisation. Also, the application can be more dynamic

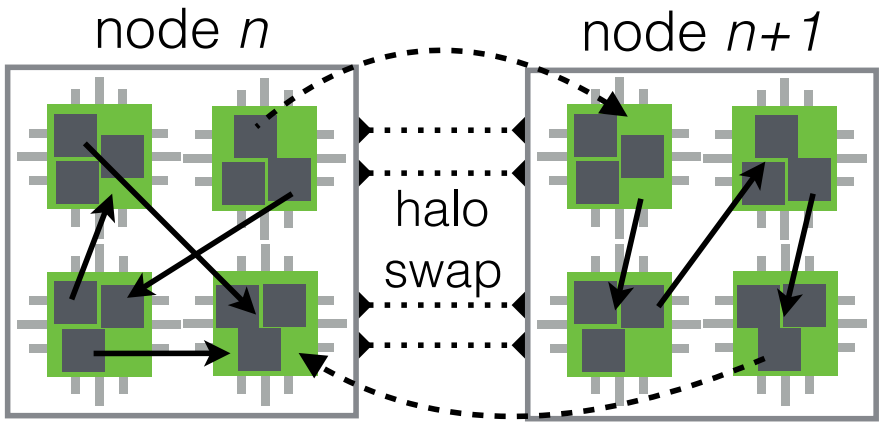


Figure 5.1: Illustration of DPLB scheme. The solid arrows show PLB within node and the dashed arrows show work movement across nodes.

because there is no need to match specific sends and receives. Some two sided communication still exists, but only where the message matching does not interfere with asynchrony. A pseudocode description of our distributed asynchronous Jacobi with DPLB implementation can be found in Appendix A. On node we use the same implementation as described in 4.1.1, thus the code is written in mixed mode with MPI and OpenMP.

Information gathering about work progress of nodes is done using a reduction implemented using remote memory access (RMA) operations. Every node publishes a small data structure containing the average progress of its problem subdomains. Other nodes can query these structures with a get operation when global balance is being checked. We note that this method may present a scalability challenge. In the future, it would be interesting to evaluate a "gossip" based communication strategy, which has in the past been used to efficiently balance a synchronous iterative application at large scale [80]. In a gossip strategy, underloaded nodes send their state information to a random set of other nodes, which then augment the received messages and pass them to a different random set of nodes and so on, until the information reaches overloaded nodes with high probability. Given the approximate nature of this method, it may be even better suited to asynchronous algorithms. However, in order to validate our approach, this implementation caveat is acceptable.

An important part of the implementation is moving subdomains between nodes dynamically and adjusting communication targets. To ensure scalability, it is important to avoid introducing a global bottleneck here, for example by using a centralised table of physical subdomain locations. Instead, in our implementation subdomains keep

track of just their neighbours' locations. When a subdomain moves, it leaves behind a message with its new host rank (i.e. MPI rank). When its neighbours perform a halo exchange, as part of the halo they also receive the message that the subdomain has moved and what the new rank is that should be queried for the desired halos.

The main component facilitating this interaction is metadata appended to halos, specifically an ID and owner rank. Upon retrieval of a halo, the metadata is checked to make sure it is as expected (initial locations of subdomains are known). If the metadata rank is not the same as the rank the halo was received from, the halo and associated subdomain have moved (the rank that does the moving changes the halo metadata to reflect the rank to which it has migrated). Once the new rank is known, an array of halo displacements is retrieved from the target rank. The array is searched to find the physical memory location of the target halo. The halo can now be retrieved and the ID checked to make sure they are correct. Only the communicating neighbours were involved in this transaction, which makes it scalable.

This extension of PLB retains the same level of applicability as the original balancing algorithm. All code-level additions are mainly to facilitate data movement between nodes and the principle of achieving load balance over time, not instantaneously, remains. Thus IPLB and DPLB can be applied to other asynchronous iterative algorithms where there is scope for splitting the problem domain and moving it between computing units, for example the Schwarz method or stochastic gradient descent.

5.2 Experiments

As our test application we continue to use Jacobi's algorithm applied to the diffusion problem in 2 dimensions. See Section 4.1.4 for details. The main difference is that the problem domain is distributed across nodes in 1 dimension, along the x axis; on-node the distribution remains 2-dimensional.

We used the HPC system Cirrus [5] for our experiments. Hardware and compiler details are as listed before in Table 4.1 except the MPI library was upgraded to Intel MPI 17.0.2.

Workload imbalance is higher in the distributed memory setting than the shared memory setting due to the relative cost of on-node and off-node communication. Also, we have increased the noise injection from affecting a single core to affecting CPU sockets or whole nodes, to further stress the system. Noise is generated by running an additional background thread that sleeps and busy-waits for set amounts of time.

Additionally, the workers' niceness[1] is set to a high value so that they have lower priority, thus yielding to the noise generating threads when active.

We chose a noise level of 40% per CPU socket (i.e. the CPU effectively runs at 60% of its normal clock frequency). This value is the mean of worst case clock frequency variations due to manufacturing variability observed in [28] when limiting node power – a factor to consider in future exascale systems with global power constraints. Also, Chunduri et al. [38] report application runtime variability between 1.18× and 1.74× (38% on average) related to network congestion on a production system. For experiments where we slow down a whole node, we chose the same level to make comparisons between experiments more direct. This can happen if both sockets are slow, the node is hot from a previous job or if there is significant network congestion.

Each experiment was repeated 5–10 times on different sets of nodes. Where possible, a series of experiments with differing settings (e.g. normal, normal plus balancer, normal plus balancer plus noise etc.) were repeated on the same node set so that differences between the experiments would be mainly due to algorithmic differences, instead of node conditions.

In *time to solution* (TTS) experiments the application runs until the global l_2-norm of the residual normalised by its initial value reaches a threshold. We set this threshold at 10^{-3}. Generally the threshold is smaller in real applications, however here we wanted to limit the total execution time and focus on performance metrics rather than the final solution. The residual calculation is done every 1000 iterations to reduce its overhead. Additionally, it is carried out asynchronously; each process computes the residual across local WIs and then issues a non-blocking Allreduce operation to combine the local residuals into the global residual. For the other experiments, the termination criterion was that one thread completes 10^4 domain updates.

We use two different problem sizes in our evaluations: 300×300 cells per core and 1000×1000 cells per core. We refer to the 300^2 problem as "small" and the 1000^2 problem as "large". These names do not reflect size in any particular application domain, but rather we observed a change in behaviour of the balancing algorithm using these values. On one node the cores are arranged in a 6×6 grid; with more nodes (and thus more cores) the global problem size scales up proportionally:

$$\text{y size} = 6 \cdot \text{y size per core} \tag{5.1}$$

$$\text{x size} = 6 \cdot \text{x size per core} \cdot \text{num nodes} \tag{5.2}$$

[1]The niceness number is used by the OS to schedule multiple processes requesting the same CPU resources. Processes with high niceness yield to processes with low niceness.

5.3 Evaluations

In this Section we present an experimental evaluation of IPLB and DPLB. Our results are broken down into different sections, building up load balancing complexity and amount of noise in the system. We present the outcomes using a range of plots which we explain first, before moving on to discussing the results.

Figures 5.2, 5.3 and 5.5 show how iteration rate and halo staleness changes across synchronisation and balancing methods. The *y* axis shows how many iterations the most stale halo was lagging behind during the run. Note that this is not the same as "update spread" which was used in the shared memory case. The most stale halo metric is more suited to asynchronous computation in a distributed memory setting because it gives a measure of staleness throughout an experiment with only point-to-point communications. The *x* axis shows the global average iteration rate normalised by the number of nodes. The best methods will be in the lower right hand corner, i.e. the aim is to both minimise staleness and maximise iteration rate.

Figures 5.6 to 5.9 show time to solution results. Less time and smaller variance is better. In addition, Figures 5.7, 5.8 and 5.9 include Tables of iteration rates (in units of 1000s of iterations per second per node) and staleness (most stale halo encountered). Figures where the *y* axis start significantly above zero are emphasised with bold numbering.

The different synchronisation types are denoted using the following labels:

ssync(i) halos from neighbouring domains must be within *i* iterations of the updating cell, otherwise update is stalled until others catch up

async(n) totally asynchronous version with each domain subsplit into *n* subdomains

Load balancing types are specified by:

s(f) CPU sockets are balanced independently every f seconds (s stands for split)

j(f) CPU sockets are balanced together every f seconds (j stands for joint)

h(f, n) CPU sockets are balanced independently every f seconds and together every nth balancing event (h stands for hybrid)

D(f) balancing across nodes is attempted every f seconds (D stands for Distributed)

IPLB independent PLB, indicated by presence of s(f), j(f) or h(f, n)

DPLB distributed PLB, indicated by presence of D(f)

The specific settings of PLB parameters were chosen to be the same as in 4.1.4 because these were found to give good performance. To reduce the number of different variants to show, we exclude sync because PLB cannot be applied to it and we also exclude ssync(10) because it is affected by noise more than ssync(30).

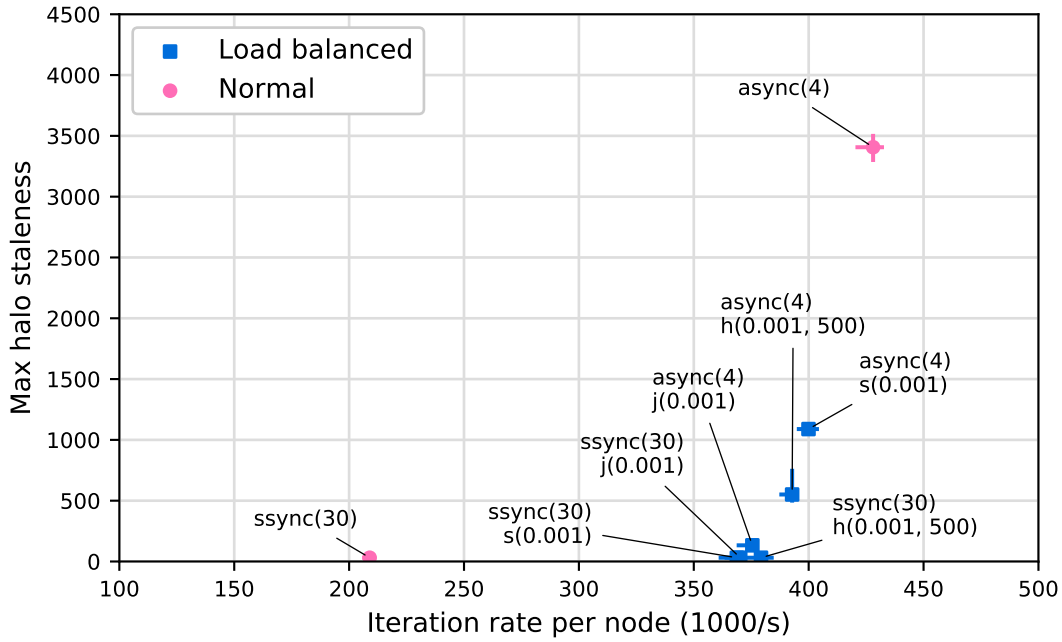
5.3.1 IPLB

We first evaluate to what extent independent instances of PLB running on each node can mitigate performance variation. Figure 5.2 shows a 2 node example with inherent imbalance (top) and with added 40% noise (bottom) to the CPU socket that is assigned the middle of the problem domain. The inherent imbalance is mainly due to increased cost of communicating halos across node boundaries. In this scenario IPLB shows the ability to reduce progress variation globally for the asynchronous version, albeit with a small decrease in iteration rate, and to significantly increase iteration rate for the semi-synchronous version. A slow socket can be mitigated to a lesser extent, as shown by the smaller staleness reduction in the "added noise" plot.

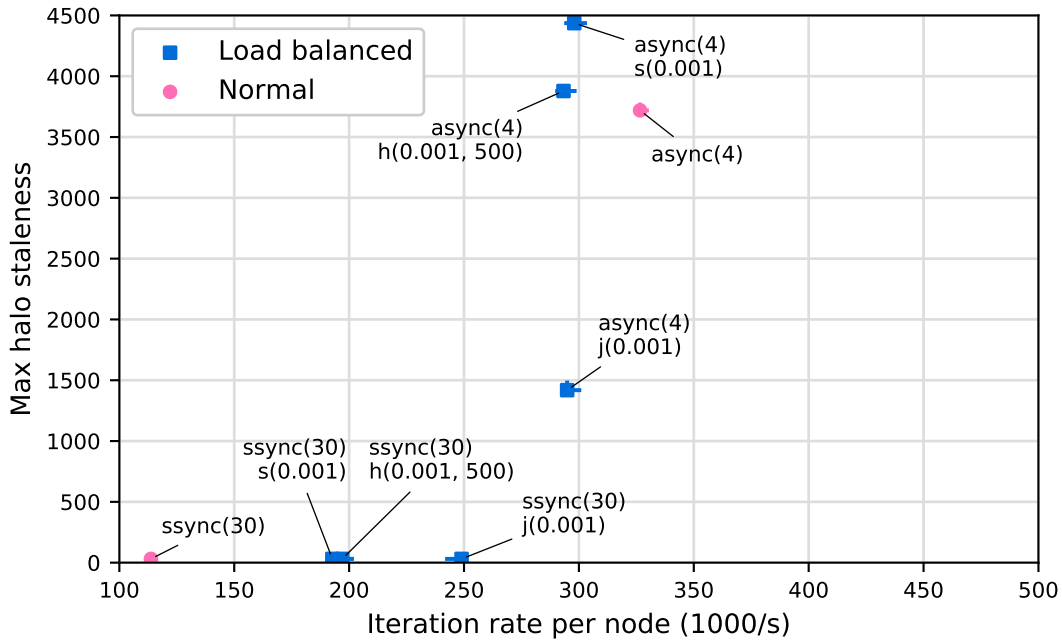
Figure 5.3 shows a 5 node version of the same experiment. Here the IPLB scheme does not reduce global progress imbalance because in this case there is more imbalance between nodes than within nodes, even with added noise.

The qualitative difference between the 2 and 5 node experiments can be seen visually in Figure 5.4. It shows snapshots of update progress at the end of a set of runs. The x and y axes correspond to coordinates of subdomains in the solution space. The vertical axis shows how many iterations have been completed for the subdomains. The closer the surface is to a flat plane, the more even progress is. When running on 2 nodes there is a progress dip near the node boundary due to the added communication load (Fig. 5.4a). Both nodes have the same dip; the imbalance is symmetric. Therefore instances of PLB running on the two nodes separately can bring both nodes to a similar level, thus reducing global staleness (Fig. 5.4b). Now consider the same experiment on 5 nodes. The middle nodes have more progress imbalance than those on the edge because the middle nodes have two neighbours, while the edge nodes only have one (Fig. 5.4c). The imbalance is symmetric for middle nodes together and the edge ones together, so the two sets can be brought to similar levels with IPLB. However, the two sets are asymmetric to each other, so globally the system is not balanced (Fig. 5.4d).

We note that in both of the above cases the semi-synchronous version of the code

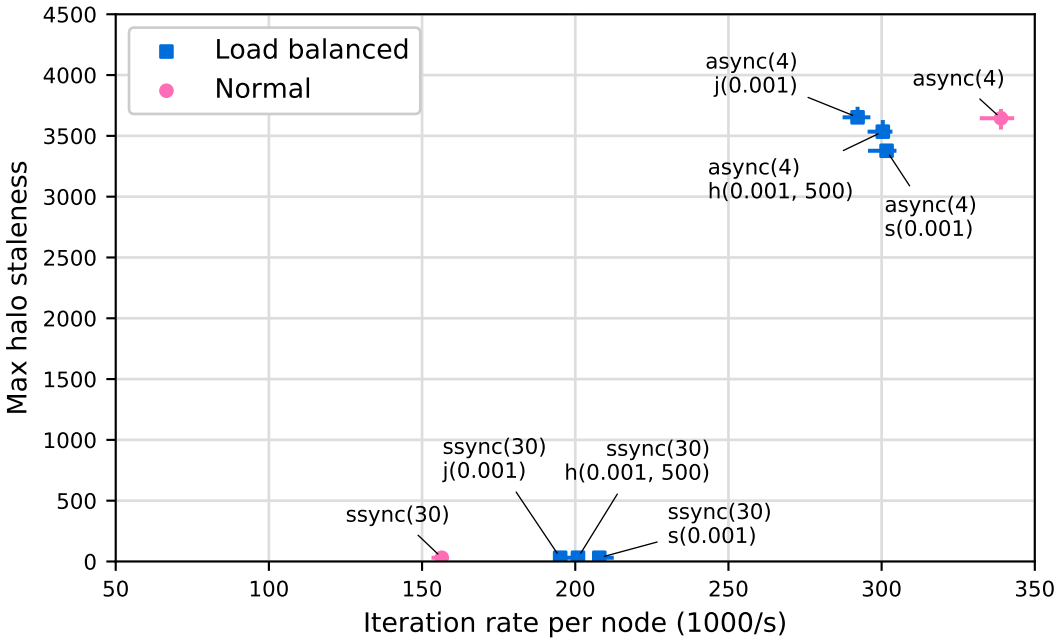


(a) No added noise

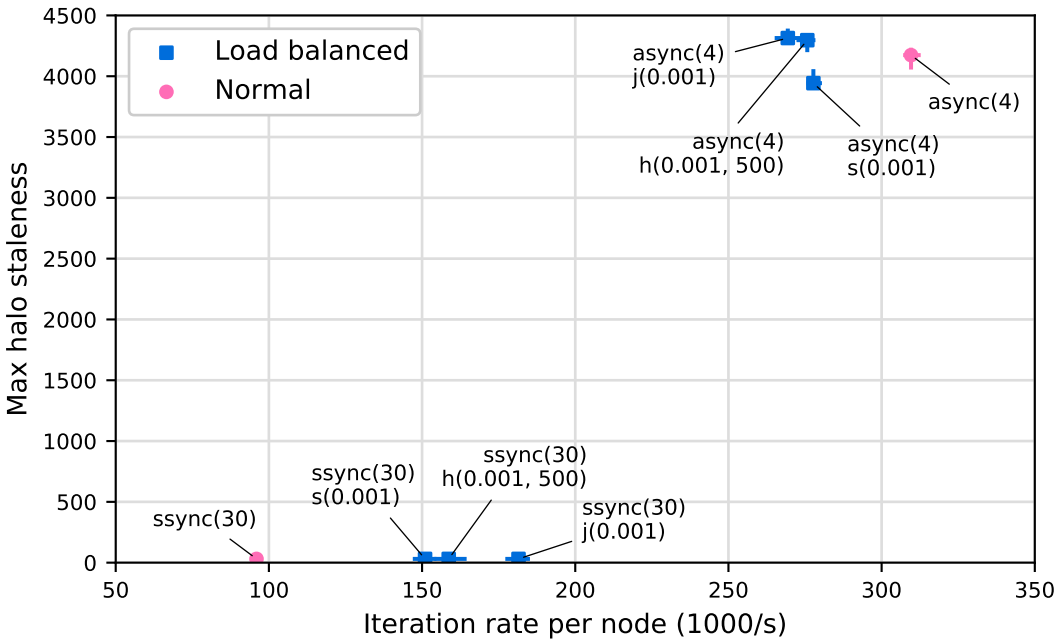


(b) With added noise

Figure 5.2: Comparison of synchronisation types for the small problem size on 2 nodes. The points represent median values and the error bars show the 25th and 75th percentiles. Closer to the lower right corner is better.

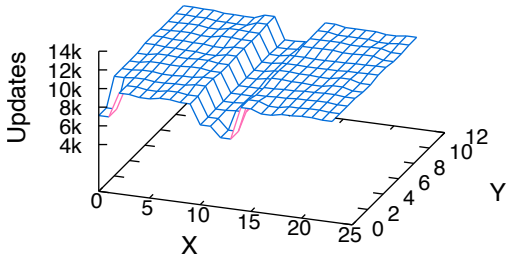


(a) No added noise

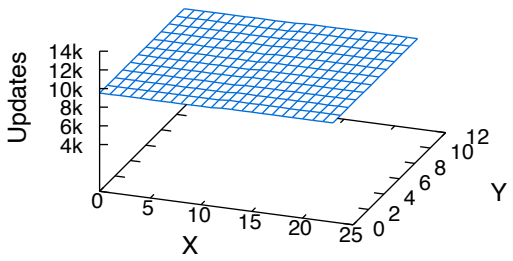


(b) With added noise

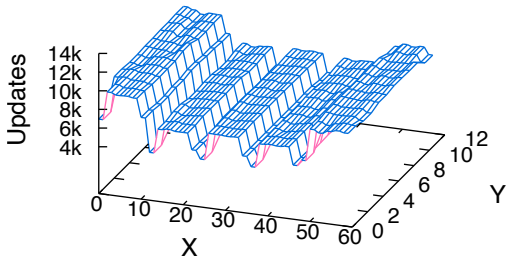
Figure 5.3: Comparison of synchronisation types for the small problem size on 5 nodes. The points represent median values and the error bars show the 25th and 75th percentiles. Closer to the lower right corner is better.



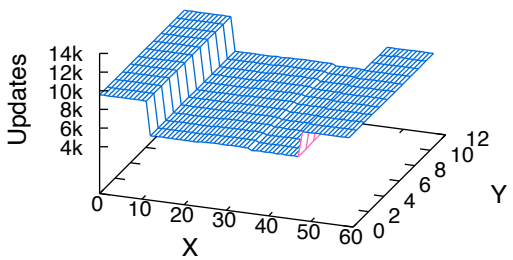
(a) async, 2 nodes



(b) IPLB, 2 nodes



(c) async, 5 nodes



(d) IPLB, 5 nodes

Figure 5.4: Surface plots of number of updates performed (vertical axis) by each problem subdomain (at x,y coordinates; horizontal axes) comparing the effect of IPLB on 2 and 5 nodes. Small problem size is used and there is no added noise. “k” stands for “thousand”.

experiences a boost in iteration rate. Without balancing, the system eventually reaches a defined staleness bound and starts running at the rate of the slowest components. When PLB is added, the noise is spread out over all available cores on each node. As a result of this smoothing, the system runs at the rate of all (slow and fast) components *averaged* over the whole node. This result would be even more dramatic in the case of a single slow core in two ways: a very small portion of the machine has a relatively larger effect on the whole system, and the noise from a single slow core can be spread out more effectively across the node.

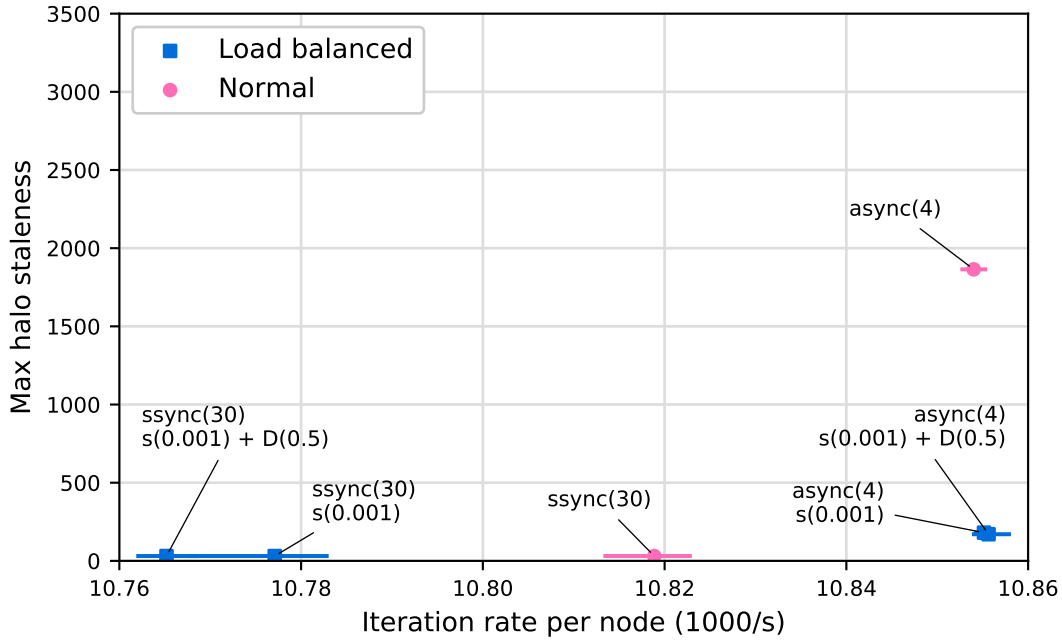
We also note that the split load balancing policy gives better staleness reduction and iteration rate on 5 nodes in most cases. For this reason we will present only this variant of PLB for the remainder of the Chapter to improve clarity of the presentation. In real world applications one may wish to fine tune the settings of PLB to a greater extent.

Next, consider Figure 5.5, which shows results for a larger problem size on 5 nodes. In this case it can be seen that IPLB can once again reduce global progress variation for the asynchronous version, even though the 5 node case has asymmetric node communication imbalance. Increasing the problem size reduces the relative cost of cross node communication in comparison to computation on node, because in 2D the perimeter of a subdomain grows linearly with the length of the side of the subdomain while the area grows quadratically. This makes the communication asymmetry less pronounced and IPLB becomes effective.

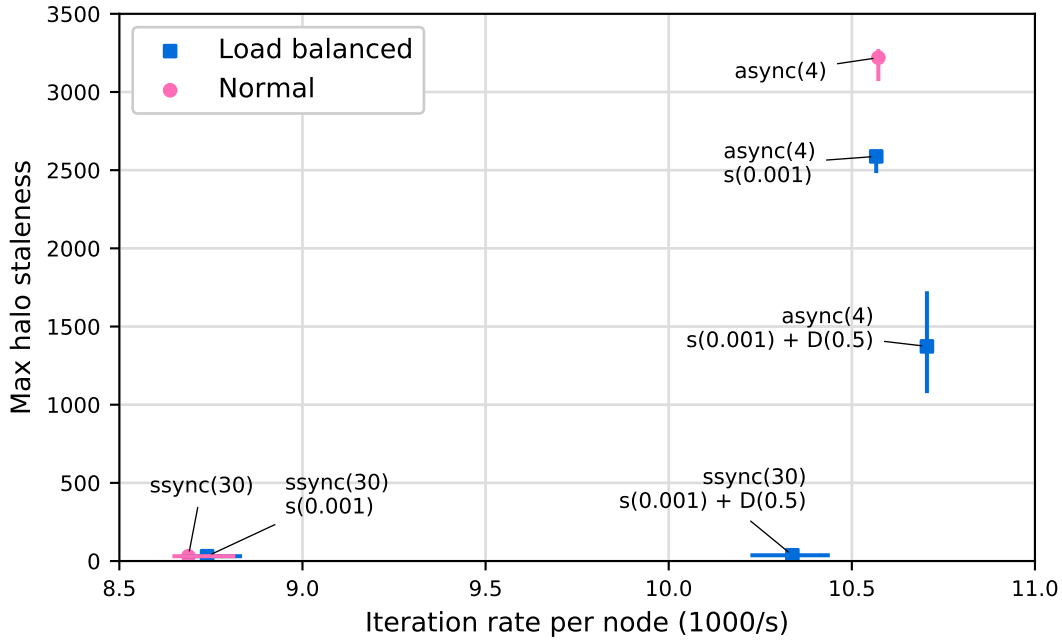
When noise is added, IPLB still reduces staleness, although to a lesser extent. Now the node with the slow CPU socket is the limiting factor instead of communication load imbalance. This again increases imbalance asymmetry between the nodes. Previously (Fig. 5.3) adding noise had a less noticeable effect because the imbalance due to communications had similar magnitude to the added noise.

The described effects of IPLB can also be observed when considering time to solution. Figure 5.6a shows TTS for the small problem size on 5 nodes. The asynchronous version does not change significantly with the addition of IPLB, while the semi-synchronous version converges faster and mitigates the added noise better. With the large problem (Fig. 5.6b) IPLB reduces TTS slightly for the asynchronous version. The semi-synchronous version experiences some slowdown, likely due to balancing overhead.

We have seen that balancing using IPLB, it is possible to smooth out variations on each node individually. If the variation between nodes is smaller than the varia-



(a) No added noise



(b) With added noise

Figure 5.5: Comparison of synchronisation types for the large problem size on 5 nodes. The best performing versions of IPLB and DPLB are shown. The points represent median values and the error bars show the 25th and 75th percentiles. Closer to the lower right corner is better. Note that the x axes have different scales.

tion within nodes, this method can reduce variation globally. The overall effect on performance is problem dependent, and there is further room for improvement.

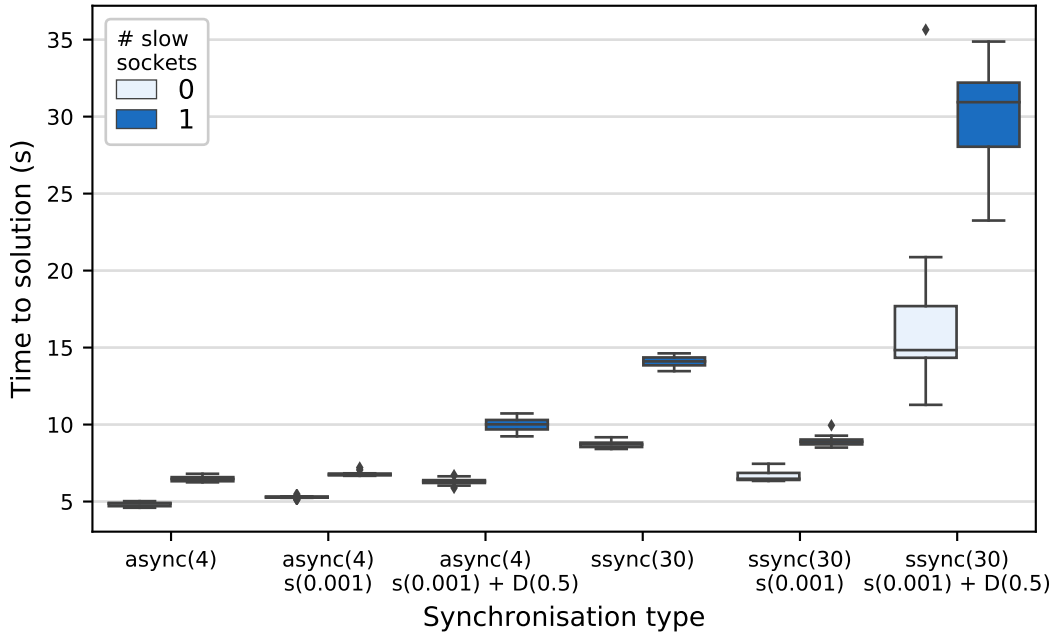
5.3.2 DPLB

To address the situations where it was not possible to reduce global progress variation, we next consider the DPLB scheme. We repeat the 5 node experiments from before, but now with the addition of cross node balancing. In Figure 5.5 it can be seen that, without added noise, DPLB performs approximately the same as IPLB. However, when a socket is noisy, DPLB reduces staleness in the asynchronous version and increases iteration rate in the semi-synchronous version.

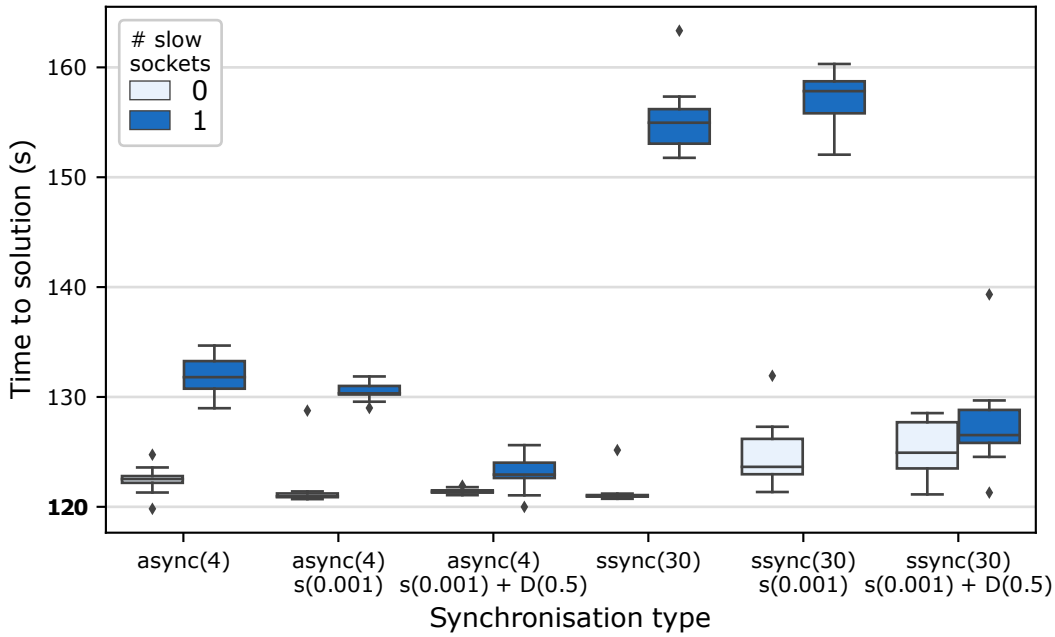
Figure 5.6 shows how time to solution is affected. For the large problem (Fig. 5.6b) DPLB results in the quickest convergence and best noise mitigation. In contrast, for the small problem size (Fig. 5.6a) the communication versus computation ratio is too high (i.e. there is insufficient work); this limits performance. Consequently, for all the remaining results shown here we use the larger, and also more realistic, problem size.

To further test the method, we increase the difficulty of the balancing problem by running on 15 nodes and also growing the number of nodes with a slow CPU socket. In order to survey the range of possible noise scenarios, a portion of the experiments has noise placed randomly and another portion has noise placed at purposely chosen locations. For the latter we picked “worst case” and “best case” noise placement, based on the problem that is being solved. The initial conditions put a Gaussian shaped source in the middle of the problem domain, so updates in the middle contribute more towards reducing the residual than the edges. Thus we add noise to components that are initially responsible for the middle of the problem domain to get worst case performance and add noise to edges to get best case performance.

Figure 5.7 shows results for the semi-synchronous version. Without balancing, the time to solution gradually increases; we also observed instances of 200%–260% slowdown when 6, 7 or 8 sockets were noisy. DPLB mitigates the noise noticeably for all noise counts, and avoids the large outliers at higher noisy socket counts. Since progress imbalance is capped, the performance difference comes from DPLB sustaining a higher iteration rate. The balanced version’s median TTS is reduced by 3–10%, except for the noiseless case where the unbalanced version is 5% faster on average. We note that the current implementation allows the staleness bound to be overstepped slightly due to subdomain updates occurring while some subdomains are being trans-

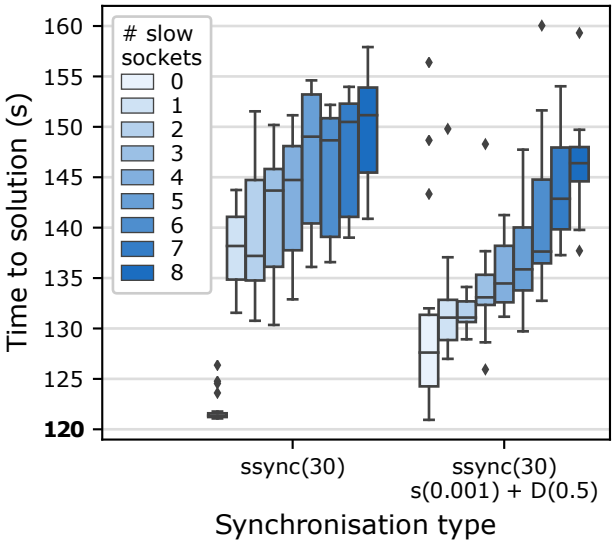


(a) Small problem size, 5 nodes.



(b) Large problem size, 5 nodes.

Figure 5.6: Comparison of IPLB and DPLB time to solution with two different problem sizes. Within each synchronization type category the left, light-blue boxplot corresponds to 0 CPU sockets running 40% slower, and the right, dark-blue boxplot corresponds to 1 CPU socket running 40% slower.



	ssync(30)		ssync(30) + DPLB	
# slow	stale	iter rate	stale	iter rate
0	31	10.80	34	10.75
1	31	9.86	39	10.44
2	31	9.34	39	10.38
3	31	9.25	39	10.30
4	31	9.20	40	10.18
5	31	8.97	40	10.01
6	31	9.02	39	9.96
7	31	8.90	40	9.70
8	31	8.76	39	9.49

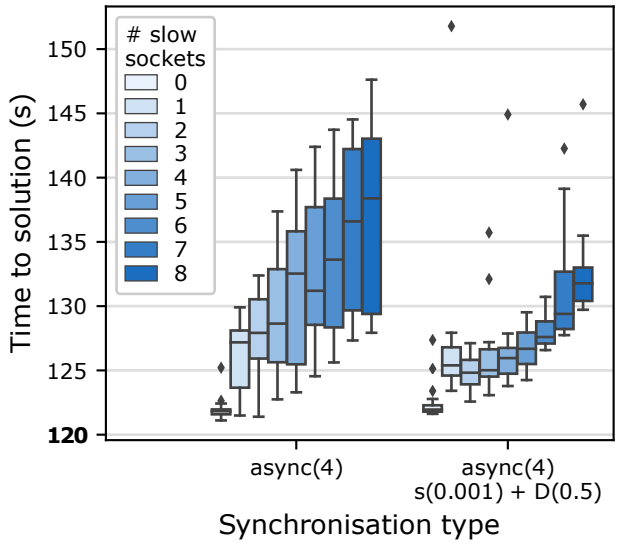
Figure 5.7: Effect of DPLB on semi-synchronous Jacobi running on 15 nodes. Color indicates the number of CPU sockets running 40% slower. The Table shows median values.

ferred between nodes.

Results for the asynchronous version can be seen in Figure 5.8. In the Table it can be seen that iteration rate is not affected adversely by DPLB and halo staleness is reduced 1.08×–1.61×. As a result, the balanced asynchronous version converges quicker for every noise setting, with a median reduction of up to 6%. The TTS of the balanced version is larger than that of the noiseless case, but this is to be expected even with perfect balancing since slow components take away the total amount of available compute power in the system. Furthermore, the worst case TTS grows at a higher rate without DPLB, which implies reduced scalability. With DPLB the worst case TTS remains mostly flat until noise is added to 5 or more sockets.

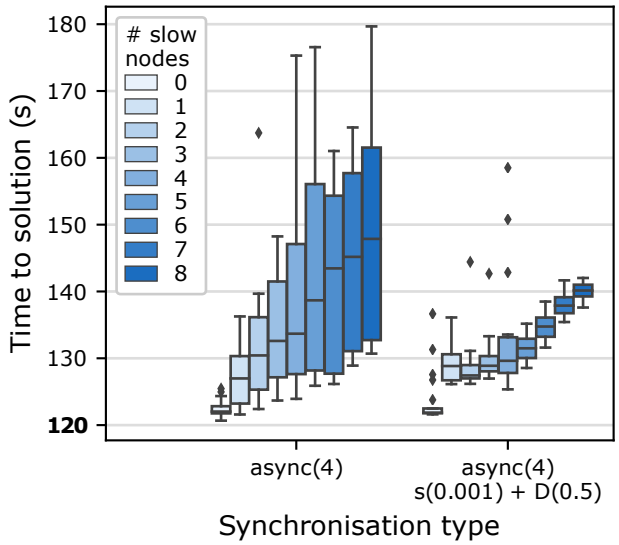
Because the asynchronous version shows better performance than the semi-synchronous version overall, we test it further by slowing down whole nodes, not just individual CPU sockets. This can occur if a job is assigned an overheating node or if there is a lot of network communication from other jobs going through the node's links. These results can be seen in Figure 5.9. The overall patterns are similar to the previous case, but more pronounced. Balancing reduces median TTS by up to 6% again, but the reduction in worst case TTS is significantly higher, as is the reduction in staleness at 1.24×–4.05×.

An important feature to emphasise is the excellent reduction in performance vari-



	async(4)		async(4) + DPLB	
# slow	stale	iter rate	stale	iter rate
0	1860	10.85	175	10.86
1	3122	10.75	2587	10.78
2	3339	10.65	2070	10.75
3	3493	10.55	2732	10.69
4	3628	10.45	2507	10.62
5	3593	10.35	2725	10.53
6	3657	10.26	2876	10.45
7	3723	10.16	3455	10.33
8	3731	10.06	3152	10.22

Figure 5.8: Effect of DPLB on asynchronous Jacobi running on 15 nodes. Color indicates the number of CPU sockets running 40% slower. The Table shows median values.



	async(4)		async(4) + DPLB	
# slow	stale	iter rate	stale	iter rate
0	1880	10.85	181	10.86
1	3794	10.65	3051	10.64
2	4755	10.46	1272	10.53
3	4617	10.26	1257	10.45
4	5377	10.07	1782	10.29
5	5591	9.87	1680	10.13
6	5575	9.68	1559	9.90
7	5581	9.47	1377	9.67
8	5709	9.33	1707	9.47

Figure 5.9: Effect of DPLB on asynchronous Jacobi running on 15 nodes. Color indicates the number of nodes running 40% slower. The Table shows median values.

Table 5.1: Runtime variability ratios comparing versions without DPLB versus with DPLB. Numbers greater than 1 show that DPLB has reduced runtime variability.

noisy	ssync socket	async socket	async node
0	0.06	1.14	8.12
1	1.21	1.86	1.47
2	4.00	2.42	3.50
3	2.19	3.55	3.88
4	1.81	4.24	6.26
5	1.03	3.39	7.65
6	0.83	4.37	5.06
7	0.89	1.51	5.71
8	1.71	3.41	11.07
extremes	1.11	1.51	2.89

ability due to DPLB. Table 5.1 shows, for each noisy component count, the ratio of the unbalanced version's spread of TTS (distance between the boxplots' whiskers) against that of the balanced version. The last line of the Table shows this ratio applied to the spread of TTS across all counts of noisy components, i.e. between the highest top whisker and lowest bottom whisker in each category. The change for the semi-synchronous code varies between $0.06\times$ (the balanced version is more variable) and $4.00\times$ (the balanced version is less variable). However, for the asynchronous code, balancing always reduces variance, ranging from $1.14\times$ to $11.07\times$. If the number of noisy components is not set at any particular value, the balanced versions exhibit between $1.11\times$ and $2.89\times$ less variation. This increased consistency in runtime is crucial for time sensitive applications, e.g. operational weather forecasting which must complete within a certain time frame [32]. It is also important in cases such as application scheduling on shared compute resources, benchmarking and keeping within budget of HPC resources.

As a final test, we ran our code on 100 nodes (3600 cores) with highly variable noise settings from run to run in order to simulate a real life scenario. For each individual run we selected a random set of nodes to be noisy; the size of the set was also chosen randomly between 0 and 15. The level of slowdown on each node in the set

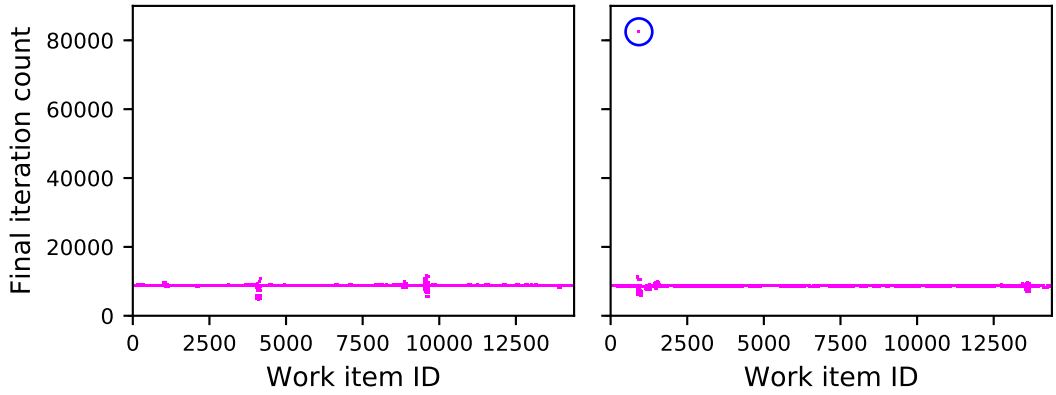


Figure 5.10: Outlier example in the 100 node DPLB runs. The left image shows a normal run and the right image shows a run with a single outlier (circled) out of 14400 work items. In both cases 5 nodes are noisy.

was chosen randomly between 15% and 40%. We performed 42 runs with the asynchronous Jacobi code and another 42 with asynchronous Jacobi plus DPLB. In 6 of the DPLB runs we observed a a single subdomain with an extreme iteration lead over all other subdomains (an example can be seen in Figure 5.10). The cause is likely a missed corner case in the implementation, since the issue does not appear in simulations. Out of an abundance of caution we discarded the pairs of runs with outliers entirely.

Runtime results can be seen in the first row of Table 5.2. Both versions performed very similarly. As described in Section 4.1.4, the Jacobi method was chosen as the test application in part due to its reliable convergence. The present experiment suggests that a corollary of this property is resistance to noise when the global problem size is large. This is evidenced by the low deviation in runtime of the version without balancing despite a large range of noise intensity. Therefore, adding load balancing in this case did not reduce runtime further because the progress variation was already absorbed by Jacobi's inherent resilience (but other input problems can be more sensitive to noise). Nevertheless, since the addition of DPLB does not increase runtime, this experiment demonstrates that DPLB has no significant overhead in this setting and it scales well.

Staleness results can be seen in the second row of Table 5.2. The most stale halo is very similar between the two versions, when averaged across experiment runs, but the best and worst case runs have lower staleness with DPLB. Hence these load balancing settings can achieve significant spread reduction in this scenario, but further tweaking of the parameters would be needed to achieve consistency.

On the whole, the results of the asynchronous algorithm with DPLB show greatly

Table 5.2: Runtime and staleness comparison on 100 nodes.

		mean	min	max	std. dev.
time to solution (s)	async	118.0	116.2	120.9	1.3
	async + DPLB	118.0	115.2	120.7	1.4
most stale halo	async	4,475.5	1,472.0	11,760.0	2,805.1
	async + DPLB	4,545.8	318.0	9,141.0	2,237.3

reduced variance in TTS and variability in update progress of problem subdomains. In addition, the worst case noise scenario TTS is less when DPLB is added while the best case noise scenario is slightly higher. These observations taken together indicate that smoothing noise is beneficial in the majority of the time. While reducing progress imbalance occurring in a less critical part of the problem domain results in a small increase in TTS, *not* reducing imbalance in a more critical part results in a much larger increase in TTS. On average, the risk of excessive runtime and progress imbalance of an asynchronous algorithm can be noticeably reduced with DPLB.

5.3.3 Profiling

To understand the overheads of the method we performed detailed profiling. The analysis was done using the asynchronous version of the code and without adding any noise to the system. DPLB was set up to perform balancing continuously, by setting the inter-node imbalance tolerance to 0. We compare DPLB against IPLB instead of the unbalanced case because doing so shows the impact of off-node balancing in particular. While IPLB does not move work between nodes, it can still change the number of threads that are performing off-node communication by moving border domains between threads. The result is increased congestion at the NIC.

Table 5.3 shows a summary of DPLB overheads in terms of time. In all cases, the time spent in DPLB related functions (querying global balance and moving subdomains) accounts for less than 1% of the total runtime. This highlights that the load balancing work is light and domain movement can be overlapped with computation. The time spent on intra-node balancing is 0.3% or less. The final column in Table 5.3 shows the total overhead of DPLB compared to IPLB. DPLB incurred an overhead of less than 3% in all cases, which indicates a bound for the minimum expected impact of staleness on runtime, at which point applying the presented load balancing techniques

Table 5.3: Load balancing overheads.

	nodes	DPLB (% of runtime)	IPLB (% of runtime)	DPLB vs IPLB runtimes (% change)
small prob.	5	0.8	0.3	+2.6
	10	0.9	0.3	+0.1
	15	0.9	0.3	+2.0
big prob.	5	<0.1	0.1	+0.2
	10	<0.1	0.1	+0.0
	15	<0.1	0.1	-0.1

Table 5.4: Data movement profiling.

	nodes	DPLB vs IPLB remote halo data (% change)	DPLB data movement (MB/thread/s)	total remote data movement (MB/thread/s)
small prob.	5	+6.8	0.361	2.315
	10	+2.4	0.181	2.215
	15	+1.8	0.121	2.197
big prob.	5	+83.6	3.874	4.157
	10	+41.1	2.112	2.356
	15	+24.8	1.286	1.510

becomes beneficial. This includes the already discussed direct overheads as well as indirect ones, e.g. the introduction of additional remote halo communications caused by moving a domain away from a node where all neighbours are local.

Table 5.4 shows a summary of data movement within the application. It can be seen that for the large problem there is a significant increase in amount of remote data movement. There is more data movement due to remote halo exchange and subdomain movement makes up most of the total data movement. However, the absolute values in terms of bandwidth are small and well within hardware limits. Additionally, all data movement metrics in Table 5.4 are falling with increasing number of nodes. This is due to an increase in the number of threads while keeping the DPLB domain movement

frequency constant. In some scenarios the load balancing frequency might have to be increased, but in our case we have already shown that the chosen settings are sufficient.

Overall, while DPLB increases inter-node data movement, this imposes only a small overhead in terms of time and scales favourably with the number of nodes. It is worth noting that because interconnects can be optimised for different message sizes and communication patterns, the overheads of DPLB may vary on different machines. However our profiling indicates that on a system with a high performance network DPLB is unlikely to have a negative impact on scaling.

5.4 Summary

We have presented two methods for applying progressive load balancing to an asynchronous algorithm in a distributed memory setting. One method, IPLB, aims to achieve global progress balance by running independent instances of PLB on each node. This approach was shown to reduce imbalance when it is present symmetrically across nodes. A second method, DPLB, and its implementation was also presented. It addresses the limitations of IPLB by adding periodic movement of work between nodes.

Evaluation of DPLB showed that, given a sufficiently large problem size, it is able to mitigate system performance variation, where IPLB can not, by a reduction of 1.08×–4.05× in global progress imbalance, 1.03×–1.10× in median time to solution and by 1.11×–2.89× in time to solution variability. We did not observe any significant overheads even when running on 100 nodes with the large problem size.

In the next Chapter we will expand the evaluation of DPLB applied to Jacobi by testing its performance in the presence of real performance variation. Additionally, we will investigate load balancing another algorithm – asynchronous stochastic gradient descent.

Chapter 6

Extended use cases of DPLB

In this Chapter we expand the evaluation of DPLB beyond simulated noise scenarios by testing on a machine experiencing real hardware-caused performance variation. We also evaluate progressive load balancing in the context of stochastic gradient descent (SGD). This algorithm is widely used in machine learning and can be run asynchronously, however using stale values reduces statistical efficiency. We investigate whether balancing the progress of learners would lead to convergence rate similar to that of synchronous SGD while maintaining the hardware efficiency advantage of asynchronous methods.

6.1 DPLB in the presence of real performance variation

At the start of 2020 unusual behaviour was noticed on 12 nodes of Fulhame, an HPC machine hosted at EPCC [81]. The CPUs in the nodes in question were running more than 4 times slower than the base clock. Most nodes were running at 2.2GHz (the expected frequency) but the slow ones were running at 0.5GHz. This was first noticed while a user was running a distributed version of the STREAM [82] benchmark which examines the distribution of single node STREAM measurements and the results came back with 12 nodes as extreme outliers. It is not clear how long the nodes had been in this state; the issue was found only due to this particular benchmarking taking place and may have been affecting jobs for many days. Eventually the problem was diagnosed as a partial failure of power supply units attached to the nodes. It is interesting that the CPUs ran at reduced speed rather than failing, and this is a good example of a real world scenario where performance variability manifested itself.

Before the nodes were repaired, we were able to run experiments to evaluate DPLB

Table 6.1: "Fulhame" test system details.

System type	HPE Apollo 70
CPU Sockets	2
CPU	ARM Marvell ThunderX2
Number of Nodes	64
Core count per CPU	32
Clock	2.2 GHz
Interconnect	Mellanox InfiniBand
Topology	Non-blocking fat tree
RAM per CPU	64 GB DDR4
Compiler	GCC 9.2
MPI library	OpenMPI 4.0.2 with MultiThreading support
Main compilation flags	-O2

in a real life setting where the noise was not artificial. Note that there was a very short window (about 2 days) of opportunity, between the diagnosis and the fix of the problem, to adapt the code to a new system and to gather data. Hence, we could not perform an exhaustive set of experiments.

Details of the machine can be found in Table 6.1. The code for Jacobi and load balancing were configured largely the same as the experiments on Cirrus (see Section 5.2), but without simulated noise. Also, the nodes on Fulhame had been set up in SMT-4 (simultaneous multithreading with 4 threads per core) mode which resulted in a different thread numbering pattern than on Cirrus. For ease of porting, we restricted the experiments to using 32 of the 64 available cores, i.e. socket 0 only. We used the large problem size (1000×1000 cells per core) and used 8 cores in the y direction and 4 in the x direction per node. The experiments terminated when the relative residual reached 10^{-3}.

The total number of normal and noisy nodes (52 and 12 respectively) on Fulhame allowed us to examine 3 scales: 5, 15 and 50 nodes. At each scale we kept the proportion of noisy to total number of nodes at 20–22%. Each run was repeated 3–5 times while keeping the placement of the noisy nodes within the problem domain the same. As a reminder, the global problem domain is split across nodes in one dimension. On 5 and 15 nodes the placement was nearly in the middle of the domain, i.e. the patterns were 3n-1s-1n for 5 nodes and 8n-3s-4n for 15 nodes where *n* stands for normal node

and *s* stands for slow node. On 50 nodes it was not possible to repeat this placement because of the location of noisy nodes so the noise was mostly on the edge nodes, either 35n-4s-5n-7s or 35n-4s-4n-6s.

6.1.1 Results

The following plots show the results of the experiments on Fulhame while it was affected by slow nodes. Each entry on the x axis shows a different configuration and within those are 3 clusters corresponding to the 3 scales. Each marker is the result from one run and the black cross shows the mean of that set.

The configurations are:

async baseline asynchronous, no load balancing, only good nodes

async asynchronous, no load balancing, 20% nodes slow

async plb asynchronous, only on-node balancing, 20% nodes slow

async dplb asynchronous, on-node and cross-node balancing, 20% nodes slow

sync baseline synchronous, no load balancing, only good nodes

sync synchronous, no load balancing, 20% nodes slow

ssync(30) semi-synchronous with staleness bound 30, no load balancing, 20% nodes slow

The results are also summarised in Table 6.2.

6.1.1.1 Runtime

The summary of time to solution results can be seen in Figure 6.1. The baseline performance for the asynchronous and synchronous variants are similar. However, with slow nodes present the synchronous variant is affected much more than the asynchronous one, slowing down nearly by the same amount that the nodes are slowed down by (around 4×). The semi-synchronous variant is affected by the noise similarly to the synchronous variant, except at 50 nodes. At the larger scale the effects of the slow nodes takes longer than the total runtime to propagate across all nodes.

It is also worth noting that the synchronous variant takes longer to complete on average as the scale increases, even though the slow node proportion and total number

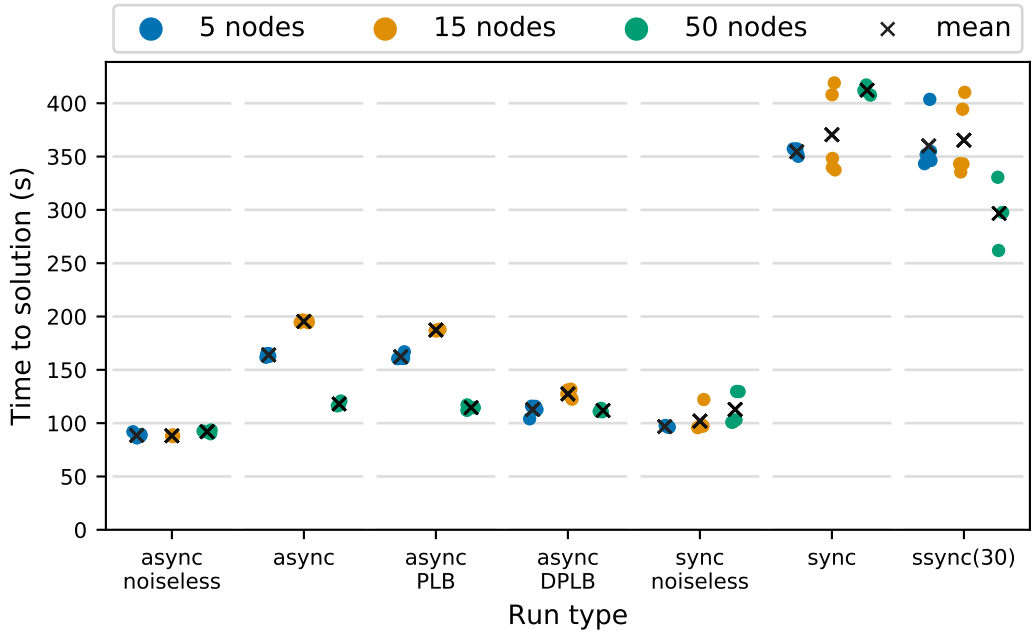


Figure 6.1: TTS comparison across different run types of Jacobi and node counts in the presence of slow nodes on Fulhame.

of iterations to converge stays about the same. The effect is also visible in the baseline synchronous runs. This average increase in runtime is due to more performance outliers at the larger scales. We examined this and found that the performance differences are down to more time spent in communications, possibly due to network congestion. This is an example of the compounding nature of noise in a synchronous system and the benefits of asynchronous computing (trials are clustered around the mean). In addition, the semi-synchronous trials have performance outliers within categories as well.

The asynchronous runs with slow nodes exhibit variable runtime across scales, which is partially explained by noise placement (it is placed in more noise-sensitive parts of the domain at 5 and 15 node scales). PLB alone does not reduce time to solution; this is expected since the noise is affecting whole nodes and not individual cores in this case. DPLB, on the other hand, significantly reduces the effect of the slow nodes. While async slows down by between 28% and 122% in comparison to the noiseless case, with DPLB this is between 22% and 44%. Note that one would not expect to get below 15% slowdown because load balancing can not recover the lost performance in the system (see Section 6.1.1.3).

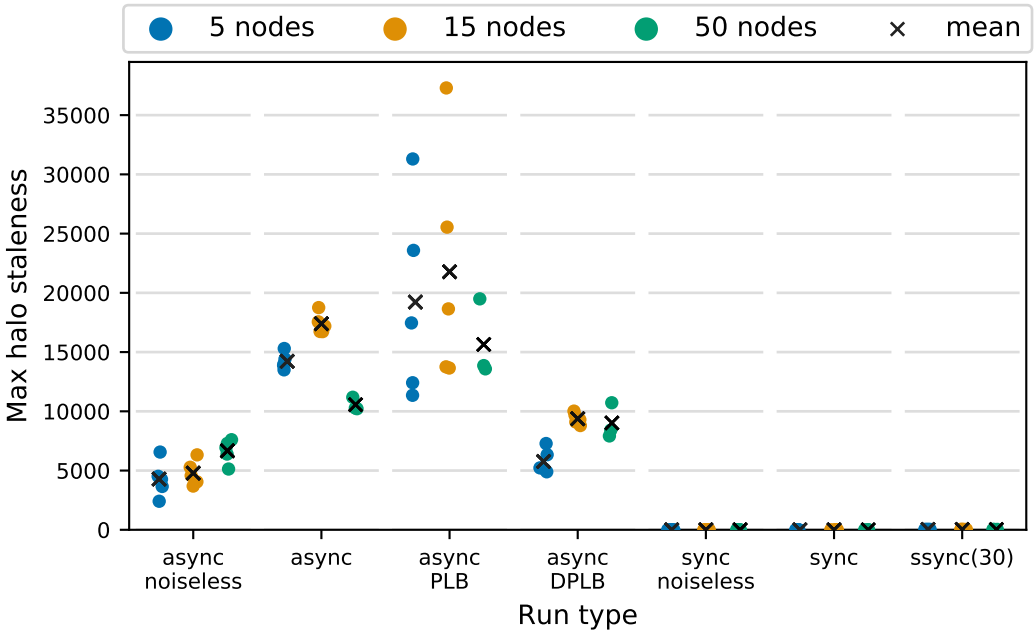


Figure 6.2: Most stale halo comparison across different run types of Jacobi and node counts in the presence of slow nodes on Fulhame. The staleness value for sync is around 1 and around 30 for ssync(30).

6.1.1.2 Most stale halo

Figure 6.2 shows the staleness results. The maximum recorded staleness values in the asynchronous cases largely matches the pattern of runtime. This indicates that staleness is growing throughout the run until convergence. However, DPLB still has reduced staleness induced by the slow nodes. The relative effect is quite small at 50 nodes, because the rate of load balancing (a tuneable parameter) was kept the same while the number of slow nodes has increased.

The exception is PLB with a wide range of staleness values, sometimes more and sometimes less than the plain asynchronous case. Since the balancing is performed within nodes, these results must be due to the differences between elements that exchange halos only on node and those that exchange halos across nodes. Load balancing in this case can make the edge elements (inter node communications) progress closer to those in the bulk (intra node communications). On the other hand, this can also create additional stress on the NICs as more threads participate in off node communication, though this could be mitigated by bundling these messages.

To consider staleness another way, one can look at the rate of staleness, i.e. maximum staleness divided by time to solution (see Figure 6.3). It can be seen in this plot

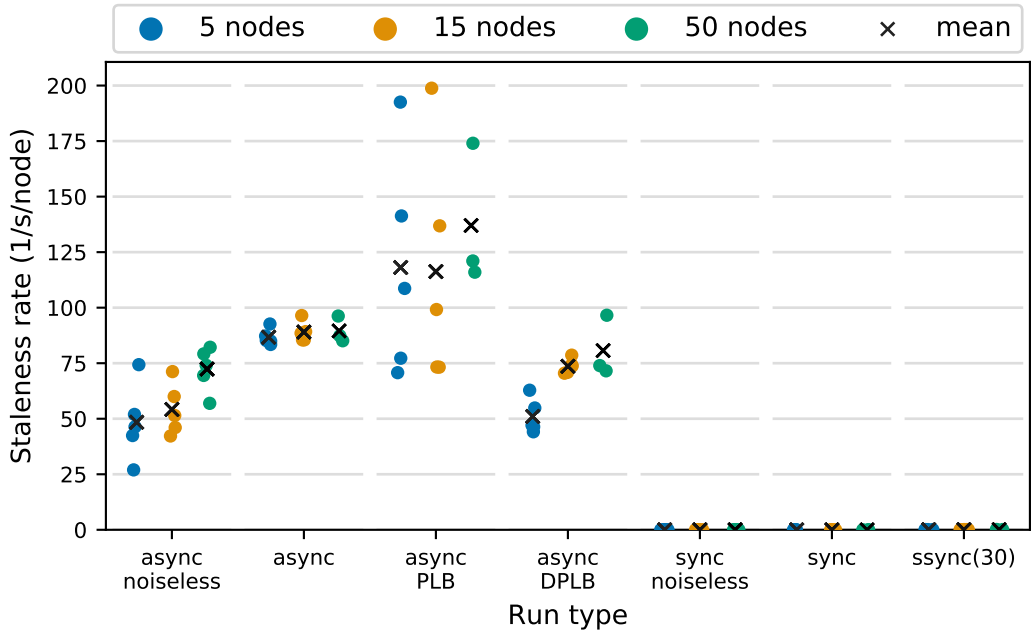


Figure 6.3: Staleness rate comparison across different run types of Jacobi and node counts in the presence of slow nodes on Fulhame. The staleness rate for sync and ssync(30) is less than 1.

that the rate of staleness for the noisy asynchronous case is the same across scales, even though the maximum staleness is different. The runtime differences are then due to noise placement, which increases time needed to converge in some runs, and so the maximum staleness increases proportionally. On the other hand, staleness rates are lower with DPLB active rather than inactive, and in some cases they are close to the baseline rates. This means that maximum staleness has been reduced by more than would be accounted for by reducing runtime by changing which nodes are affected by noise.

6.1.1.3 Iteration rate

The average iteration rate for asynchronous runs is very stable (see Figure 6.4). The drop in performance when the noisy nodes are present is about 15%, which matches expectations given that about 15% computing power is lost when 20% of nodes are running at about 25% (0.5GHz over 2.2GHz) of their normal clock frequency. Equation 6.1 shows the calculation that was performed to get the percentage of lost performance; note that node homogeneity under noiseless conditions is assumed. PLB and DPLB do not affect the overall iteration rate significantly, which is an indication of the

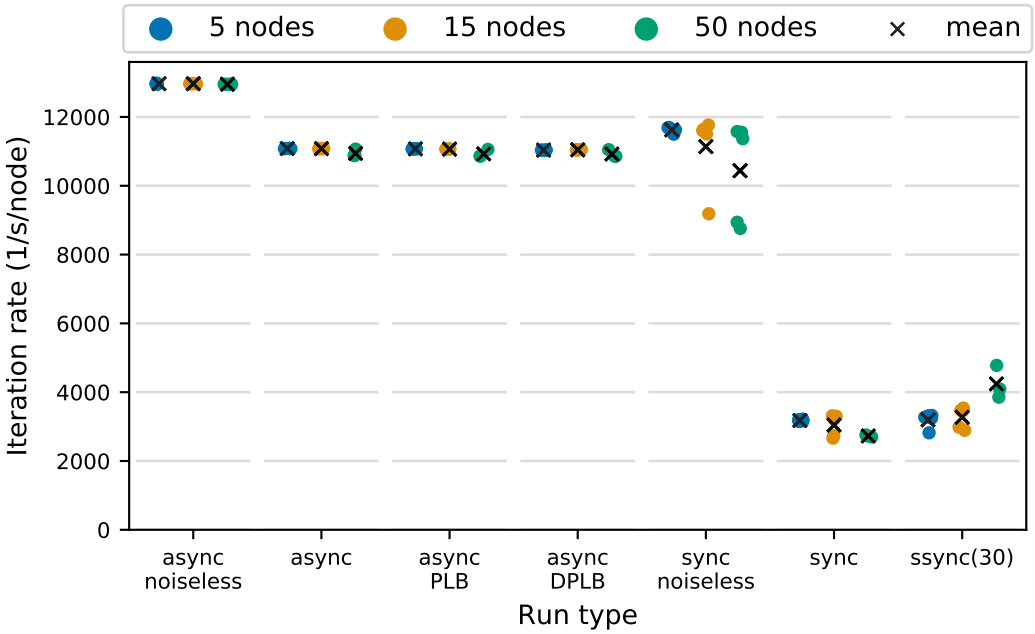


Figure 6.4: Iteration rate comparison across different run types of Jacobi and node counts in the presence of slow nodes on Fulhame.

low overhead of the method.

$$\begin{aligned} \text{lost performance} &= 1 - \sum \text{fraction of nodes} \cdot \text{node performance} \\ &= 1 - (\frac{4}{5} \cdot 1 + \frac{1}{5} \cdot \frac{1}{4}) = 15\% \end{aligned} \tag{6.1}$$

The pattern of iteration rates of the synchronous and semi-synchronous cases match runtime differences observed before.

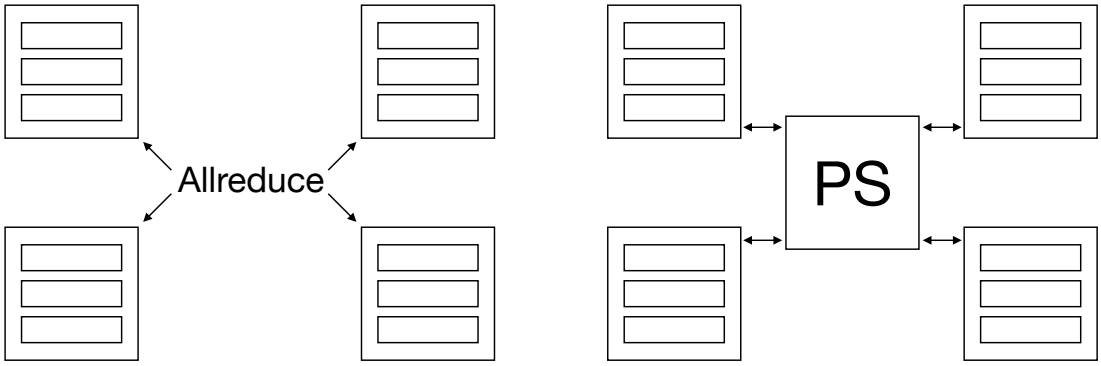
Table 6.2: Summary table of measurements on Fulhame. Mean values only.

	Run type / # nodes	async noiseless	async	async PLB	async DPLB	sync noiseless	sync	ssync(30)
Time to solution (s)	5	88.5	164.0	162.3	112.8	96.8	354.5	360.1
	15	88.2	195.4	187.3	127.4	102.0	370.5	365.3
	50	92.0	118.0	114.5	111.9	112.9	412.2	296.7
Max staleness	5	4,281.4	14,223.2	19,222.2	5,769.2	2.0	2.0	31.0
	15	4,783.8	17,394.0	21,779.4	9,376.6	2.0	2.0	31.0
	50	6,667.6	10,562.0	15,645.0	9,021.7	2.0	2.0	31.0
Staleness rate (1/s/node)	5	48.4	86.7	118.1	51.0	0.0	0.0	0.1
	15	54.2	89.0	116.3	73.6	0.0	0.0	0.1
	50	72.4	89.6	137.0	80.7	0.0	0.0	0.1
Iteration rate (1/s/node)	5	12,967.1	11,082.8	11,072.9	11,037.1	11,623.6	3,177.9	3,194.6
	15	12,967.0	11,080.3	11,066.1	11,045.8	11,139.5	3,044.7	3,271.8
	50	12,952.0	10,941.3	10,928.0	10,922.8	10,440.1	2,727.6	4,241.8

6.2 DPLB applied to Stochastic Gradient Descent

Mathematical optimisation encompasses a multitude of methods concerned with finding the set of inputs which provide the optimal output, for example the parameters of a function which describe the minimum or maximum of the function. These methods are widely used in machine learning to refine the parameters of a model which is used to perform some task like image classification. A popular class of methods is called *gradient descent* which explores the space of model parameters by following the gradient at each point until a minimum is reached. A widely used algorithm in this class is stochastic gradient descent (SGD) which improves upon the earlier batch gradient descent method by producing model updates based on individual or small groups (mini-batches) of training examples instead of the entire training dataset. Progress towards the solution is more erratic, but this can help in avoiding local optima [83].

Two common ways to achieve parallelisation and distribution of SGD are illustrated in Figure 6.5. The first method (6.5a) relies on data parallelism to split mini-batches between workers and synchronises at each model update. In the second method (6.5b) each worker produces their own model updates with a potentially stale view of the system. Asynchronous SGD (ASGD) is normally implemented using a parameter server [10]. The server stores the coefficients of the learning model while other nodes compute updates to the model, send these updates to the parameter server and



(a) Synchronous SGD **(b)** Asynchronous SGD

Figure 6.5: Two methods of distributing SGD.

receive the latest version of the model. In this context staleness is the number of updates to the model that have happened between two reads from the parameter server by a worker. Issues with using excessively stale updates include reduced accuracy of the learned model and instability [46]. For these reasons one may opt to limit the asynchrony in a system by employing a hybrid scheme [57] where the worker nodes are split into groups. Synchronous computation is used within groups and asynchronous computation via parameter servers is used between groups.

If a slow node is encountered inside a group, that group will run at the rate of the slow node. This in turn decreases the rate of reading from the parameter server so the group will be computing updates using more stale values than other groups. That can have a negative effect on convergence and the learned model may have systematic inaccuracies if the group handles a different subset of the training data, as indeed needs to be the case when training on very large datasets.

A straightforward way to apply PLB principles in this use case is to periodically move the slow node's assignment between different groups. In this way the effects of staleness are spread across all groups and are not concentrated on one group. An implementation can use similar tactics to the ones used by DPLB to detect excessive staleness and ensure that the load is balanced over time.

An important point to note is that each group of nodes needs to own a different set of training data for this simple implementation of DPLB to have an effect. When the slow node is part of a group, that whole group runs slower due to synchronisation within the group. If all groups have the same training data they are virtually identical, so moving the slow node around (and consequently slowing down different groups) does not change anything in the global picture. On the other hand, if groups have

different parts of the training data, DPLB style load balancing spreads staleness evenly. This results in all data contributing in an equal manner, thus likely resulting in a better final model.

6.2.1 Setup of experiments

We evaluated the application of DPLB to SGD using the Caffe machine learning framework [84]. We chose Caffe because the Intel distribution of it implements distributed hybrid ASGD using groups of synchronous workers communicating asynchronously between groups through parameter servers. Additionally, it was used in [57] to show multiple petaflop scaling of a real machine learning problem.

Caffe is a large codebase, and investigating it revealed that modifying the code to add DPLB functionality would have been a time consuming undertaking. Instead, we chose to manipulate the background noise in such a way as to emulate the presence of DPLB in Caffe. We ran a second application alongside Caffe on the worker nodes that generated background work, or noise. The background task communicates across the nodes to coordinate which nodes generate noise and which do not. In this way we can simulate work load balancing taking place without actually needing to modify the original application.

Note that here the load balancing happens in a slightly different way than in general PLB. Normally, the iteration rate in one part of the problem domain is sped up at the same time as slowing the iteration rate down in another part. By moving the noise, the iteration rate is only slowed down where the noise is present but no part is sped up. The full version can be implemented by adding noise to the whole system, which would allow removing it temporarily to speed the iteration rate in one part up and more noise could be added to slow the iteration rate down in another part of the domain. However, in the case of ASGD staleness does not accumulate in the same way as before, since we consider a read from the parameter server to reset staleness. Thus, simply moving the noise suffices to emulate DPLB.

The noise level was set to 40% as in the Jacobi experiments and the emulated DPLB load balancing took place once a second, i.e. the noise was moved from one node to the next once a second. The noise moving frequency is approximately the same as the one for the Jacobi experiments. There the frequency was once every 0.5 seconds, but that does not account for any delays in executing the load balancing, whereas here the change in balance is nearly instant.

We ran our experiments on Cirrus (see Table 4.1 for hardware details) using the CIFAR-10 dataset [85]. CIFAR-10 is a standard machine learning test dataset and its topology is verified and supported by Intel Caffe[1]. It is a relatively small dataset, so there is not enough data or work to facilitate extensive scaling. As a result, our experiments were run using 1 parameter server and 4 worker groups with 1 node (36 cores) in each group. This is sufficient since the number of groups is similar to what was used in [57] (between 2 and 8 groups) and it is the key metric that determines the level of staleness in the system [67]. Groups queue at parameter servers for model updates, so the number of groups correlates with the average length of the queue, provided that load is balanced.

The CIFAR-10 dataset is small enough to fit in memory of a single node. However, in Caffe nodes read the training data in a round robin fashion from a central database so the training examples are effectively split into non-overlapping sets between the nodes. Shuffling the database between epochs (an epoch is a single pass over the entire dataset) would alleviate this issue. However, shuffling the database comes with a performance cost and is not possible for larger datasets. For the purposes of these experiments, we turned data shuffling off in order to retain training sample heterogeneity between nodes so that the present DPLB implementation would be applicable.

We ran the following experiment configurations:

- Synchronous on 4 nodes.
- Synchronous on 4 nodes with added noise on 1 node.
- Asynchronous on 4 worker nodes with 1 server node.
- Asynchronous on 4 worker nodes with 1 server node and added noise on 1 node.
- Asynchronous on 4 worker nodes with 1 server node and added moving noise on 1 node at any one time (DPLB).

Each synchronous run was repeated 3 times and each asynchronous run was repeated 10 times. We used a fixed learning rate of 0.001, batch size of 100, momentum of 0.9 and ran for 50000 training iterations.

Table 6.3: Runtime results averaged across repetitions.

	runtime (seconds)	
	mean	stddev
Sync, 4 nodes	949	20
Sync, 4 nodes, noisy	2017	61
ASync, 4 nodes	1028	72
ASync, 4 nodes, noisy	1980	32
ASync, 4 nodes, noisy, DPLB	1186	43

6.2.2 Experiment results

Table 6.3 shows the runtime results for of each type of experiment. The synchronous method completes in least time and and its runtime is most consistent. The asynchronous baseline is slower due to communication passing through the parameter server as opposed to MPI_Allreduce based communication in the synchronous version which is more efficient at this scale.

Once noise is introduced, the runtime approximately doubles both for the synchronous and asynchronous versions. In the synchronous version, the normal nodes need to wait for the slow node at the end of each iteration. This is not the case in the asynchronous version. However, Caffe does not terminate when a target loss or accuracy is reached, but when the predefined number of iterations are complete. As a result, 3 nodes finish their calculations and then wait on the remaining slow node to complete its share of the iterations. The emulated DPLB case mitigates the effect of the noise, resulting in runtime that is consistent with losing 40% performance on one of 4 nodes.

It could be argued that the nodes which are running normally could compute additional updates while waiting for the slow node or the remaining iterations on the slow node could be dropped. However, this would have implications on the trained model from a statistical point of view and is outside the scope of this thesis.

Table 6.4 shows the results for the final values of loss and accuracy of the trained models. Loss is the objective function which is minimised during training, and accuracy is simply the fraction of correctly classified data points. Accuracy is computed

[1] "Currently, the topology for CIFAR-10 in Intel Caffe is verified with asynchronous SGD functionality" – from `github.com/intel/caffe/wiki/Asynchronous-SGD-in-Intel-Caffe`. Last accessed Jul 2021.

Table 6.4: Test loss and accuracy results averaged across repetitions.

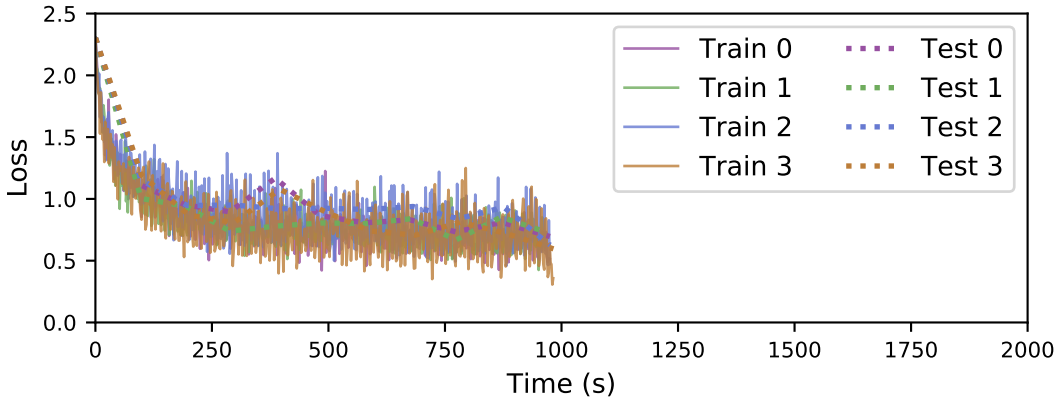
	test loss		test accuracy	
	mean	stddev	mean	stddev
Sync, 4 nodes	0.601	0.009	0.793	0.001
Sync, 4 nodes, noisy	0.607	0.018	0.793	0.008
ASync, 4 nodes	0.634	0.040	0.789	0.012
ASync, 4 nodes, noisy	0.810	0.032	0.757	0.008
ASync, 4 nodes, noisy, DPLB	0.613	0.036	0.795	0.010

using the final version of the model obtained at the end of training.

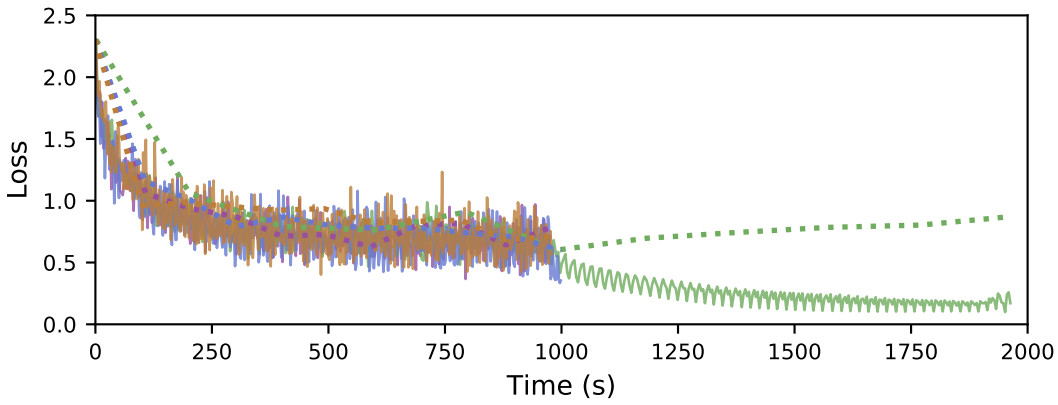
The synchronous versions achieve the lowest loss across the experiments. The asynchronous versions have higher loss on the whole, especially with a noisy node present, but DPLB reduces it significantly. Accuracy follows a similar pattern with DPLB being able to recover accuracy lost to execution noise. While the differences in accuracy appear small, any change in precision of a model can be crucial, depending on the application domain. Thus this is an important result – the noise affects both runtime and accuracy of training, but spreading the effects of the noise evenly across nodes mitigates the negative effects.

Figure 6.6 shows examples of how the loss computed on training and testing data changes during the training process of ASGD. This illustrates the runtime and model accuracy observations seen before. Figure 6.6a shows the case where all nodes are running normally and loss decreases over time as the model is trained. In Figure 6.6b a noisy node is added. Training proceeds as before until the 1000 second mark when 3 out of the 4 nodes have finished their iterations. The remaining slow node continues training the model based only on the data available to it. The training loss decreases but the test loss increases, which indicates that overfitting is taking place since there is no input from the other nodes' data to moderate training. This results in long training time and worse accuracy of the final model. If the effect of noise is distributed between nodes (Figure 6.6c), the negative model accuracy results are mitigated and runtime is increased by a much smaller amount.

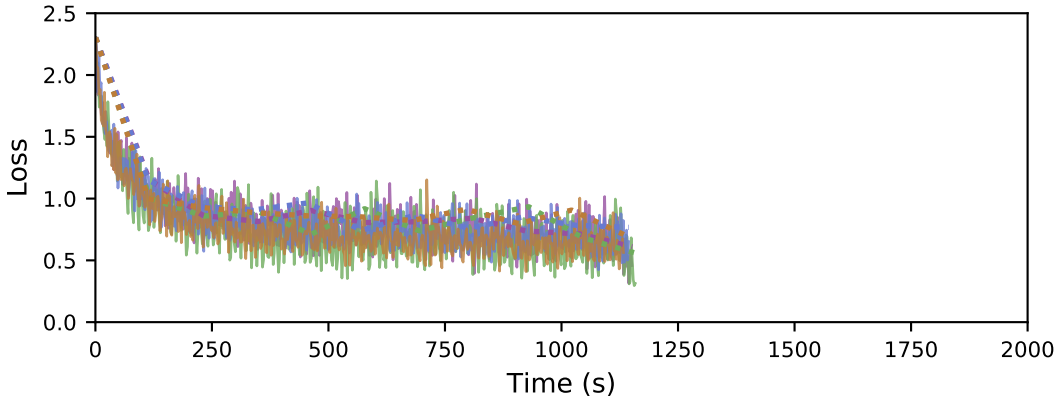
Finally, we examined the staleness of updates. In ASGD a common metric of staleness is the number of updates that have been committed to the parameter server between a write to and a read from the parameter server by a worker node. This metric resets at each read, rather than accumulating.



(a) ASGD



(b) ASGD with a noisy node



(c) ASGD with a noisy node and emulated DPLB

Figure 6.6: Intermediate ASGD loss values in three different noise scenarios. The legend is the same for all plots. Each line colour represents a different node. The solid lines show loss computed on training data and the dotted lines show loss computed on test data.

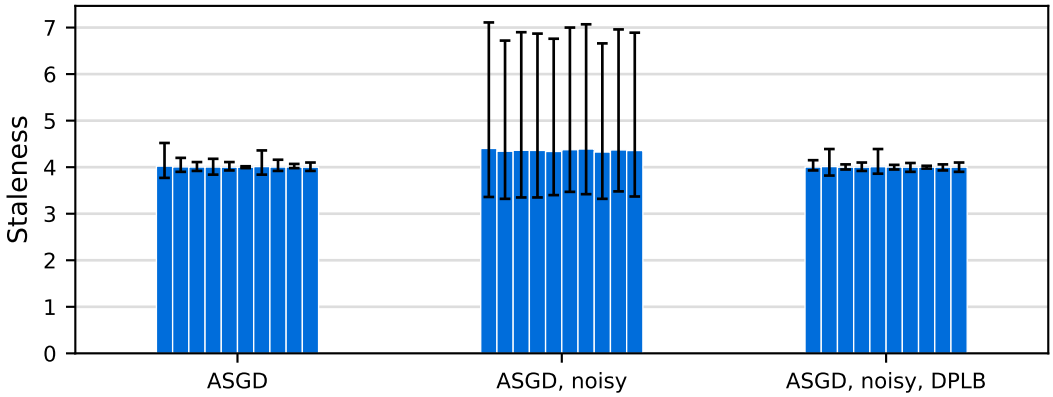


Figure 6.7: ASGD staleness measured in the middle of training on 4 nodes. Each bar is an individual run. The height shows average staleness across nodes and the error bars show the minima and maxima of average node staleness.

For each ASGD experiment we collected a sample of 10^5 staleness values after the first 10^5 values were discarded. These data correspond to approximately one minute of runtime one minute after the training had started. The samples represent the stable "bulk state" of each experiment (for illustration, see Figure 6.6). The inclusion of staleness from the whole of the run would also include the tail in the non-DPLB noisy case which would skew summary statistics for the bulk state because it would introduce many values where only one node communicates with the parameter server. It is worth noting that there are occasional spikes in staleness where individual updates are executed with model staleness on the order of hundreds. These correlate with intermediate calculations of the model's performance on the test dataset. However, these do not appear to affect the results significantly since they are very rare and affect all nodes in similar measure.

The results can be seen in Figure 6.7. Each column represents one run. Within a run we calculate the mean staleness $\bar{s}_i$ for each node i of the 4. The height of the column is the mean of these values $\frac{\sum_i \bar{s}_i}{4}$; this shows the central tendency of staleness in the run. The error bars are the minimum and maximum $\bar{s}_i$ within each set of 4; this shows the range of staleness.

It can be seen that noise increases the average staleness by a small amount. However, the noisy node consistently sees a 75% more stale model. Interestingly, the remaining nodes that are not affected by noise see a slightly less stale model. Since the noisy node commits updates less frequently, it also creates less staleness for the other nodes. Moving the noise in a DPLB fashion restores average and extreme staleness

values to the levels of the noiseless case.

6.3 Practical considerations of applying DPLB to other algorithms

The implementation of DPLB within the Jacobi application that we have been using as the running example is a good reference point for further implementations. While it is not trivial to enable DPLB within other applications, for example ASGD, it is possible to reuse much of the existing logic. Balancing decisions are done in a routine not specific to our Jacobi application, so the balancer could be abstracted out as a library in the future. A new application would need to define two functions: (i) a function to return staleness (or progress) of a work item and (ii) a function to move a work item from one core or node to another. Additionally, the application would need to have initialisation and finalisation calls for the library. Load balancing could either happen in a background thread or the application could call a library progress function regularly.

The requirement for function (i) and the different possible definitions of staleness mean that it is not possible to implement DPLB at a higher level, for example in a similar way to the OS thread manager. However, as was shown with ASGD, the background noise level can be manipulated with a pattern that imitates the presence of load balancing. This is a useful tool for quickly evaluating noise sensitivity of asynchronous algorithms and how they might respond to progressive load balancing. The main limitation of this approach is that there is no guarantee how performance variation correlates with staleness in an application. For instance, even variation in background noise is unlikely to produce a predictable pattern of staleness if workload is not distributed evenly in the first place. With knowledge of the application and experimentation a suitable noise pattern to simulate may be found.

6.4 Summary

We presented experiments showing that DPLB can mitigate performance variability in a real life scenario where noise is present in a system. It succeeded in its goal to reduce staleness while retaining high iteration rate. However, DPLB requires further parameter tuning, e.g. increased volume of domain movement, with more slow nodes

present. Also, there is a diminishing need for balancing with increased problem size for asynchronous Jacobi and overall staleness does not always have a strong effect on time to solution (confirming the observations in Section 5.3.2).

Next we evaluated the noise sensitivity of asynchronous stochastic gradient descent and its response to emulated load balancing. From our experiments we concluded that ASGD is a favourable candidate to benefit from DPLB. It reduces staleness in updates and allows all nodes to compute the requested number of data passes in equal time, even in the presence of noise. As a result, ASGD could mitigate the negative effects of noise, reducing runtime and improving accuracy. This has the potential to enable solving larger optimisation and machine learning problems where ASGD performance would suffer due to performance variability.

Finally, we briefly mapped a path how to explore and expand the application of DPLB to other asynchronous algorithms.

Chapter 7

Conclusion

In this thesis we examined the problem of performance variability in the context of iterative asynchronous algorithms, a theme that appears in many research publications on asynchronous algorithms. Time to solution of an asynchronous algorithm is a function of two components: iteration rate, which is increased due to the lack of synchronisation, and convergence rate, which is lowered due to the use of stale values. While it is common to recognise this relationship, most attention has been devoted to the former and relatively little to the latter. The increasing risk of performance variability in modern supercomputing systems makes the iteration rate advantages of asynchronous algorithms increasingly appealing and the concerns of using stale values more pressing. Therefore we embarked on creating a load balancing scheme specifically designed for asynchronous algorithms.

We first developed a simulation framework to quickly evaluate different strategies of load balancing. After a number of iterations, we identified a method which was able to handle both hardware variability and imbalanced workload. The method – progressive load balancing – works on an overdecomposed problem space and moves work items between workers in such a way to increase the iteration rate for work items that are falling behind in updates and to decrease the iteration rate for work items that are racing ahead. The goal is not to equalise iteration rates across the whole system, as in traditional load balancing, but rather to keep the number of iterations completed on each work item similar, effectively balancing the load over time. The net result is that the slowing effect of noisy components is averaged across the system.

We evaluated PLB on two systems in a shared memory setting using the Jacobi algorithm as a test case. It was found that PLB is effective in maintaining a high iteration rate while reducing update staleness, thus mitigating the impact of a noisy

core and reducing time to solution. It performed significantly better than the commonly used load balancing technique of work stealing.

In a distributed memory setting, we evaluated two methods: independent PLB and distributed PLB. In IPLB an instance of PLB is run on each node independently; this is effective in reducing staleness in situations where there is symmetric imbalance between nodes. In DPLB an instance of PLB is run on each node and also work is exchanged between nodes; this enables staleness reduction in the general case. It was found that DPLB has little overhead and it significantly reduced staleness, absolute time to solution as well as its variability.

We were also able to test DPLB in a scenario with real and severe performance variation. The results confirmed the observations made with simulated noise. Finally, we evaluated the potential of using DPLB with another application: asynchronous stochastic gradient descent. Using the machine learning framework Caffe and emulating the effects of DPLB, we showed that noise can be mitigated in terms of runtime and that the resulting model accuracy is retained or improved. These results demonstrate the real world applicability of DPLB and its potential to generalise to other iterative asynchronous algorithms.

7.1 Discussion, limitations and future work

In this thesis we have demonstrated that iterative asynchronous algorithms with progressive load balancing show promise as an important application implementation strategy on current and future supercomputing systems. Here we discuss the limitations of the presented work and identify areas that would particularly benefit from further research.

Current boundaries of PLB for improving performance

We were able to show runtime improvements up to 15 nodes but data for larger scales is limited. The cause for this is the natural noise resistance of the examined algorithm and test problem at large scale. As we increased the problem size, staleness in the system had a diminished effect on time to solution. A potential solution is to test general Jacobi solving an explicit matrix with a large range of values in it to increase sensitivity to staleness. One could even construct the matrix that does not meet the convergence criteria of Jacobi for guaranteed convergence under asynchrony (Section 2.3.2).

Nevertheless, we were able to show that the overhead of balancing is low which is essential for scaling. Also, noise affects both large and small node counts, so the developed load balancing methods remain useful. Additionally, it would be worthwhile to extend the investigation to problems with higher dimensions to better match the patterns found in many production applications. DPLB would not require any significant alterations, however increasing problem dimensionality would generally change the balance between off-node communications and computations of the application, and load balancing cost for DPLB, so this is a good area for further discovery.

A second potential constraint on the applicability of PLB is the number of iterations that the target asynchronous algorithm needs in order to converge. If an algorithm converges within a few 100s of iterations and there is a surge of imbalance, there may not be enough time to observe and mitigate staleness. Further research could ascertain typical iteration thresholds for algorithms to benefit from PLB, and identify ways in which the performance improvements of PLB can be brought to algorithms with smaller number of iterations to convergence. For example, there may be a way to break down large iterations into smaller chunks which could act as sub-iterations and could be load balanced with more agility. An interesting possible way forward is to investigate implementing asynchronous algorithms in a task-based framework, with collections of tasks that make up an iteration could be treated as these sub-iterations.

Autotuning and optimising PLB

There are multiple tuneable parameters in DPLB. The set that was used in our experiments proved rather versatile and a good default. However, on Fulhame running on 50 nodes, with 10 of those nodes experiencing 75% slowdown, the ability to reduce staleness was insufficient. The balancing settings, in particular the amount of data movement across nodes, did not provide enough bandwidth to tackle the volume of staleness. While balancing parameters can be adjusted to match this case, a better solution would be for the balancer to automatically tune itself at runtime. The tuneable parameters are not black boxes and have a clear intuition behind them, so it is a matter of quantifying their effects for an auto-tuner to make use of. Some of this work could be carried out in the simulator we used in Chapter 3.

Improvements can be made to PLB itself by taking advantage of additional information. Measurements of processor speed and the amount of work in each problem domain can be used to predict the evolution of staleness and skip unnecessary bal-

ancing steps. Information about memory hierarchy and network topology could be combined with a cost model to find compromises between balancing effectiveness and the cost of moving data as well as introducing inefficiencies in the communication pattern. Similarly to auto-tuning, it might be desirable to request a target staleness from the user, and allow PLB to find an optimal path towards it.

Theory advances needed to improve PLB

A foundational issue in the area of asynchronous algorithms is the limited understanding of the relationship between staleness and convergence rate. While outside the scope of this thesis, future work on load balancing for asynchronous algorithms would greatly benefit from theoretical research resulting in a more detailed understanding of the effects of staleness. It is likely that some amount of staleness is beneficial, and PLB has the potential to take advantage of this by generating different staleness profiles, not just minimising staleness. In the meantime, it is reasonable to aim to reduce staleness because doing so brings an asynchronous algorithm more in line with its synchronous counterpart which is known to work well.

Practical steps to encourage PLB adoption

To explore further applications of PLB, it would be beneficial to implement the balancer as a library as described in Section 6.3. Additionally, the DPLB emulation strategy (as used in Section 6.2) could be implemented as a light framework to evaluate the effects of DPLB without needing to implement it first. To improve the accuracy of the emulation, one could add staleness output from the application running in the framework and feed it back into the framework.

PLB as a way to expand use of asynchronous algorithms

There is potential that PLB will enable new asynchronous algorithms. Semi-synchronous algorithms limit staleness, but, in a noisy environment, they develop a pattern of staleness, so some parts of the problem domain would continuously consume or produce stale values. In contrast, balancing over time, as with PLB, leads to a more even distribution of staleness. As an example, consider clustering performed using k-means. Using this method, a set of datapoints are assigned to a set of centroids and the assignment is iteratively improved to minimise the distance between each datapoint and

centroid. K-means can be computed in parallel with data distributed across nodes and the set of centroids that is computed communicated either through a central server or between neighbourhoods of nodes [86]. Each update "pulls" centroids towards the locations represented by the updating node's data. A staleness pattern effectively adds more weight to some data, but load balanced asynchrony breaks the pattern and may lead to more accurate clustering. We did some preliminary work on asynchronous k-means to explore the approach for new algorithms and found evidence for the described pattern; there is scope for further investigation.

Asynchronous algorithms are not widely adopted in large part due to their radically different nature from traditional synchronous algorithms. However, asynchrony is very important in battling noise and machine learning has used asynchronous learning algorithms like ASGD to achieve petascale performance [57]. Semi-synchronous algorithms may be easier to approach than fully asynchronous ones, but their performance reverts to that of synchronous algorithms if performance variation is persistent. PLB provides the bounded staleness nature of semi-synchronous algorithms and the performance benefits of fully asynchronous algorithms. Thus PLB, in addition to enhancing the ability of asynchronous algorithms to tackle performance variability, is also a step towards broader adoption of asynchronous algorithms.

7.2 Summary

This Chapter gave a brief overview of the work carried out in this thesis and how it demonstrates that enhancing asynchronous algorithms with progressive load balancing is an effective method for tackling noise in supercomputers. It also discussed a number of limitations of PLB and areas that would benefit from further research.

My hope is that this research will encourage broader adoption of asynchronous algorithms and prove to be an important component in addressing the challenge of using supercomputers efficiently in the presence of performance variability.

Appendix A

Distributed asynchronous Jacobi with DPLB pseudocode

```
Data: T, N, threshold, halosWindow, halosRMAInfoWindow, workItems
shouldContinue ← TRUE;
MPI_Win_lock_all(..., halosWindow, ...);
while shouldContinue do
    foreach workItem in workItems do
        get_halos_RMA(workItem, halosWindow, halosRMAInfoWindow);
        update workItem;
        copy new halos of workItem into RMA accessible storage;
    end
    if T time has passed since last DPLB application then
        minR, maxR ← least and most progressed rank (via RMA);
        minR transfers random work item to maxR using non-blocking send;
    end
    if N iterations have been performed since last residual calculation then
        globalResidual ← use non-blocking reduction to find global residual;
        if globalResidual < threshold then
            shouldContinue ← FALSE;
        end
    end
end
MPI_Win_unlock_all(halosWindow);
```

```
Function get_halos_RMA(workItem, halosWindow, halosRMAInfoWindow)
    foreach neighbour of workItem do
        target ← neighbour.rank ;          // last known location of neighbour
        switch target do
            case target is local do
                look for halo in local storage;
                if halo was not found where it was expected then
                    skip neighbour ;  // use stale values, halo is in transit
                else
                    halo ← get halo from local storage
                end
            case target is a remote rank do
                MPI_Win_lock(..., target, halosRMAInfoWindow, ...);
                halosRMAInfo ← MPI_Get(..., target, halosRMAInfoWindow,
                 ...);
                MPI_Win_unlock(..., target, halosRMAInfoWindow, ...);
                look for halo on remote rank target via halosRMAInfo;
                if halo was not found in halosRMAInfo then
                    skip neighbour ;  // use stale values, halo is in transit
                else
                    offset ← get via halo entry from halosRMAInfo;
                    halo ← MPI_Get(..., target, offset, halosWindow, ...);
                    MPI_Win_flush_local(target, halosWindow);
                end
            end
        end
        haloRank ← rank found in "breadcrumb" metadata of halo;
        if haloRank not equal to target then
            neighbour.rank ← haloRank ;          // neighbour has migrated
        else
            copy retrieved halo data to workItem;
        end
    end
```

Bibliography

[1] Justs Zarins and Michèle Weiland. Progressive load balancing of asynchronous algorithms. In *Proceedings of the Seventh Workshop on Irregular Applications: Architectures and Algorithms*, IA3'17, pages 5:1–5:9. ACM, 2017.

[2] Justs Zarins and Michèle Weiland. Progressive load balancing in distributed memory. In *Proceedings of the International Conference on Parallel Computing, PARCO 2019*, volume 36 of *Advances in Parallel Computing*, pages 127–136. IOS Press, 2019.

[3] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[4] ARCHER. `http://www.archer.ac.uk`. Accessed: 01/01/2021.

[5] Cirrus. `http://www.cirrus.ac.uk`. Accessed: 01/01/2021.

[6] John Sartori, Aashish Pant, Rakesh Kumar, and Puneet Gupta. Variation-aware speed binning of multi-core processors. In *2010 11th International Symposium on Quality Electronic Design (ISQED)*, pages 307–314. IEEE, 2010.

[7] D. Chazan and W. Miranker. Chaotic relaxation. *Linear Algebra and its Applications*, 2(2):199–222, 1969.

[8] Frédéric Magoulès, Daniel B. Szyld, and Cédric Venet. Asynchronous optimized Schwarz methods with and without overlap. *Numerische Mathematik*, 137(1):199–227, 2017.

[9] Hartwig Anzt, Stanimire Tomov, Jack Dongarra, and Vincent Heuveline. A block-asynchronous relaxation method for graphics processing units. *Journal of Parallel and Distributed Computing*, 73(12):1613–1626, 2013.

[10] Stefan Hadjis, Ce Zhang, Ioannis Mitliagkas, Dan Iter, and Christopher Ré. Omnivore: An optimizer for multi-device deep learning on cpus and gpus. *arXiv preprint arXiv:1606.04487*, 2016.

[11] Dganit Amitai, Amir Averbuch, Samuel Itzikowitz, and Eli Turkel. Asynchronous and corrected-asynchronous finite difference solutions of PDEs on MIMD multiprocessors. *Numerical Algorithms*, 6(2):275–296, 1994.

[12] Gordon E Moore et al. Cramming more components onto integrated circuits, 1965.

[13] Robert H Dennard, Fritz H Gaensslen, Hwa-Nien Yu, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[14] Mark Bohr. A 30 year retrospective on Dennard's MOSFET scaling paper. *IEEE Solid-State Circuits Society Newsletter*, 12(1):11–13, 2007.

[15] Peter Bauer, Alan Thorpe, and Gilbert Brunet. The quiet revolution of numerical weather prediction. *Nature*, 525(7567):47–55, 2015.

[16] Jack Dongarra and Piotr Luszczek. *TOP500*, pages 2055–2057. Springer US, Boston, MA, 2011.

[17] Robert Lucas, James Ang, Keren Bergman, and Shekhar Borkar. DOE ASCAC subcommittee report February 10. 2014.

[18] Jack Dongarra et al. The international exascale software project roadmap. *Int. J. High Perform. Comput. Appl.*, 25(1):3–60, 2011.

[19] John L Hennessy and David A Patterson. *Computer Architecture: A Quantitative Approach; 4th ed.* Elsevier, Burlington, MA, 2007.

[20] Marc Snir, Robert W Wisniewski, Jacob A Abraham, Sarita V Adve, Saurabh Bagchi, Pavan Balaji, Jim Belak, Pradip Bose, Franck Cappello, Bill Carlson, et al. Addressing failures in exascale computing. *The International Journal of High Performance Computing Applications*, 28(2):129–173, 2014.

[21] Nosayba El-Sayed and Bianca Schroeder. Reading between the lines of failure logs: Understanding how HPC systems fail. In *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*, pages 1–12. IEEE, 2013.

[22] Keith A. Bowman, Steven Duvall, and J.D. Meindl. Impact of die-to-die and within-die parameter fluctuations on the maximum clock frequency distribution for gigascale integration. *IEEE Journal of Solid-State Circuits*, 37(2):183–190, 2002.

[23] Saisanthosh Balakrishnan, Ravi Rajwar, Mike Upton, and Konrad Lai. The impact of performance asymmetry in emerging multicore architectures. *ACM SIGARCH Computer Architecture News*, 33(2):506–517, 2005.

[24] Eric Humenay, David Tarjan, and Kevin Skadron. Impact of process variations on multicore performance symmetry. In *2007 Design, Automation & Test in Europe Conference & Exhibition*, pages 1–6. IEEE, 2007.

[25] Bharathwaj Raghunathan, Yatish Turakhia, Siddharth Garg, and Diana Marculescu. Cherry-picking: Exploiting process variations in dark-silicon homogeneous chip multi-processors. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2013*, pages 39–44. IEEE Conference Publications, 2013.

[26] Exploiting process variability in voltage/frequency control. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 20(8):1392–1404, 2012.

[27] Fabrizio Petrini, Darren J Kerbyson, and Scott Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *Supercomputing, 2003 ACM/IEEE Conference*, pages 55–55. IEEE, 2003.

[28] Yuichi Inadomi, Tapasya Patki, Koji Inoue, Mutsumi Aoyagi, Barry Rountree, Martin Schulz, David Lowenthal, Yasutaka Wada, Keiichiro Fukazawa, Masatsugu Ueda, et al. Analyzing and mitigating the impact of manufacturing variability in power-constrained supercomputing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 78. ACM, 2015.

[29] Allan Porterfield, Rob Fowler, Sridutt Bhalachandra, Barry Rountree, Diptorup Deb, and Rob Lewis. Application runtime variability and power optimization for exascale computers. In *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers*, page 3. ACM, 2015.

[30] D. Hackenberg, R. Schöne, T. Ilsche, D. Molka, J. Schuchart, and R. Geyer. An energy efficiency feature survey of the Intel Haswell processor. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 896–904, 2015.

[31] Pedro J Garcia, J Flich, J Duato, I Johnson, Francisco J Quiles, and F Naven. Dynamic evolution of congestion trees: Analysis and impact on switch architecture. *Lecture Notes in Computer Science (HiPEAC 2005)*, 3793:266–285, 2005.

[32] ARCHER. `https://www.ecmwf.int/sites/default/files/elibrary/2018/18588-met-office-hpc-update.pdf`. Accessed: 01/07/2021.

[33] Greg Faanes, Abdulla Bataineh, Duncan Roweth, Tom Court, Edwin Froese, Bob Alverson, Tim Johnson, Joe Kopnick, Mike Higgins, and James Reinhard. Cray cascade: a scalable HPC system based on a dragonfly network. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–9. IEEE, 2012.

[34] Daniele De Sensi, Salvatore Di Girolamo, Kim H McMahon, Duncan Roweth, and Torsten Hoefler. An in-depth analysis of the slingshot interconnect. *arXiv preprint arXiv:2008.08886*, 2020.

[35] Robert Underwood, Jason Anderson, and Amy Apon. Measuring network latency variation impacts to high performance computing application performance. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 68–79, 2018.

[36] Saurabh Jha, Archit Patke, Jim Brandt, Ann Gentile, Benjamin Lim, Mike Showerman, Greg Bauer, Larry Kaplan, Zbigniew Kalbarczyk, William Kramer, et al. Measuring congestion in high-performance datacenter interconnects. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, pages 37–57, 2020.

[37] J. A. Ang, R. F. Barrett, R. E. Benner, D. Burke, C. Chan, J. Cook, D. Donofrio, S. D. Hammond, K. S. Hemmert, S. M. Kelly, H. Le, V. J. Leung, D. R. Resnick, A. F. Rodrigues, J. Shalf, D. Stark, D. Unat, and N. J. Wright. Abstract machine models and proxy architectures for exascale computing. In *Proceedings of the 1st International Workshop on Hardware-Software Co-Design for High Performance Computing*, Co-HPC '14, pages 25–32, Piscataway, NJ, USA, 2014. IEEE Press.

[38] Sudheer Chunduri, Elise Jennings, Kevin Harms, Christopher Knight, and Scott Parker. A generalized statistics-based model for predicting network-induced variability. In *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 59–72. IEEE, 2019.

[39] Kurt B. Ferreira, Patrick Bridges, and Ron Brightwell. Characterizing application sensitivity to OS interference using kernel-level noise injection. In *2008 SC - International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2008.

[40] Patrick M Widener, Scott Levy, Kurt B Ferreira, and Torsten Hoefler. On noise and the performance benefit of nonblocking collectives. *The International Journal of High Performance Computing Applications*, 30(1):121–133, 2016.

[41] Mark Giampapa, Thomas Gooding, Todd Inglett, and Robert W Wisniewski. Experiences with a lightweight supercomputer kernel: Lessons learned from blue gene's cnk. In *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–10. IEEE, 2010.

[42] Sarp Oral Feiyi Wang, David A Dillow, Ross Miller, Galen M Shipman, Don Maxwell, and Dave Henseler Jeff Becklehimer Jeff Larkin. Reducing application runtime variability on Jaguar XT5. In *Cray User Group (CUG) Meeting*, 2010.

[43] Balazs Gerofi, Masamichi Takagi, Atsushi Hori, Gou Nakamura, Tomoki Shirasawa, and Yutaka Ishikawa. On the scalability, performance isolation and device driver transparency of the IHK/McKernel hybrid lightweight kernel. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1041–1050. IEEE, 2016.

[44] Sudheer Chunduri, Kevin Harms, Scott Parker, Vitali Morozov, Samuel Oshin, Naveen Cherukuri, and Kalyan Kumaran. Run-to-run variability on Xeon Phi

based Cray XC systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–13, 2017.

[45] John D McCalpin. HPL and DGEMM performance variability on the Xeon Platinum 8160 processor. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 225–237. IEEE, 2018.

[46] Jianmin Chen, Xinghao Pan, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. Revisiting distributed synchronous SGD. *arXiv preprint arXiv:1604.00981*, 2016.

[47] Kurt B. Ferreira, Patrick G. Bridges, Ron Brightwell, and Kevin T. Pedretti. The impact of system design parameters on application noise sensitivity. *Cluster Computing*, 16(1):117–129, 2011.

[48] Laxmikant V Kale and Sanjeev Krishnan. Charm++ a portable concurrent object oriented system based on C++. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 91–108, 1993.

[49] Dimitri P Bertsekas and John N Tsitsiklis. Some aspects of parallel and distributed iterative algorithms – a survey. *Automatica*, 27(1):3–21, 1991.

[50] James Demmel and Hong Diep Nguyen. Parallel reproducible summation. *IEEE Transactions on Computers*, 64(7):2060–2070, 2014.

[51] H T Kung. Synchronized and asynchronous parallel algorithms for multiprocessors. *New Directions and Recent Results in Algorithms and Complexity*, 1976.

[52] Peter K Jimack and Mark A Walkley. Asynchronous parallel solvers for linear systems arising in computational engineering. *Computational technology reviews*, 3:1–20, 2011.

[53] Diego A. Donzis and Konduri Aditya. Asynchronous finite-difference schemes for partial differential equations. *Journal of Computational Physics*, 274:370–392, 2014.

[54] Barry Lee, Stephen F. McCormick, Bobby Philip, and Daniel J. Quinlan. Asynchronous fast adaptive composite-grid methods: Numerical results. *SIAM Journal on Scientific Computing*, 25(2):682–700, 2003.

[55] Asynchronous fast adaptive composite-grid methods for elliptic problems: Theoretical foundations. *SIAM Journal on Numerical Analysis*, 2006.

[56] Janis Keuper and Franz-Josef Pfreundt. Asynchronous parallel stochastic gradient descent: A numeric core for scalable distributed machine learning algorithms. In *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments*, page 1. ACM, 2015.

[57] Thorsten Kurth, Jian Zhang, Nadathur Satish, Evan Racah, Ioannis Mitliagkas, Md. Mostofa Ali Patwary, Tareq Malas, Narayanan Sundaram, Wahid Bhimji, Mikhail Smorkalov, Jack Deslippe, Mikhail Shiryaev, Srinivas Sridharan, Prabhat, and Pradeep Dubey. Deep learning at 15PF: Supervised and semi-supervised classification for scientific data. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, pages 7:1–7:11. ACM, 2017.

[58] Evgenia A. Kontoleontos, Varvara G. Asouti, and Kyriakos C. Giannakoglou. An asynchronous metamodel-assisted memetic algorithm for CFD-based shape optimization. *Engineering Optimization*, 44(2):157–173, 2012.

[59] Douglas V. de Jager and Jeremy T. Bradley. Extracting state-based performance metrics using asynchronous iterative techniques. *Performance Evaluation*, 67(12):1353–1372, 2010.

[60] Dimitri P Bertsekas and John N Tsitsiklis. *Parallel and distributed computation: numerical methods*. Prentice-Hall, Inc., 1989.

[61] Keval Vora, Sai Charan Koduru, and Rajiv Gupta. Aspire: exploiting asynchronous parallelism in iterative algorithms using a relaxed consistency based dsm. In *ACM SIGPLAN Notices*, volume 49, pages 861–878. ACM, 2014.

[62] Joo Hwan Lee, Jaewoong Sim, and Hyesoon Kim. Bssync: Processing near memory for machine learning workloads with bounded staleness consistency models. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 241–252. IEEE, 2015.

[63] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. Maiter: An asynchronous graph processing framework for delta-based accumulative iterative computation. *IEEE Transactions on Parallel and Distributed Systems*, 25(8):2091–2100, 2013.

[64] Dimitri P. Bertsekas and John N. Tsitsiklis. Some aspects of parallel and distributed iterative algorithms—a survey. *Automatica*, 27(1):3–21, 1991.

[65] I. Bethune, J. M. Bull, N. J. Dingle, and N. J. Higham. Performance analysis of asynchronous jacobi's method implemented in MPI, SHMEM and OpenMP. *International Journal of High Performance Computing Applications*, 28(1):97–111, 2013.

[66] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 693–701, 2011.

[67] Ioannis Mitliagkas, Ce Zhang, Stefan Hadjis, and Christopher Ré. Asynchrony begets momentum, with an application to deep learning. In *Communication, Control, and Computing (Allerton), 2016 54th Annual Allerton Conference on*, pages 997–1004. IEEE, 2016.

[68] Shuxin Zheng, Qi Meng, Taifeng Wang, Wei Chen, Nenghai Yu, Zhi-Ming Ma, and Tie-Yan Liu. Asynchronous stochastic gradient descent with delay compensation. In *International Conference on Machine Learning*, pages 4120–4129, 2017.

[69] Dganit Amitai, Amir Averbuch, Samuel Itzikowitz, and Eli Turkel. Asynchronous and corrected-asynchronous finite difference solutions of PDEs on MIMD multiprocessors. *Numerical Algorithms*, 6(2):275–296, 1994.

[70] James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Gregory Ganger, Garth Gibson, Kimberly Keeton, and Eric Xing. Solving the straggler problem with bounded staleness. In *Proceedings of The 14th Workshop on Hot Topics in Operating Systems*, 2013.

[71] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B. Gibbons, Garth A. Gibson, Greg Ganger, and Eric P. Xing. More effective distributed ML via a stale synchronous parallel parameter server. In *Advances in Neural Information Processing Systems*, pages 1223–1231, 2013.

[72] Henggang Cui, James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Abhimanu Kumar, Jinliang Wei, Wei Dai, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. Exploiting bounded staleness to speed up big data

analytics. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, 2014.

[73] Hartwig Anzt, Stanimire Tomov, Jack Dongarra, and Vincent Heuveline. A block-asynchronous relaxation method for graphics processing units. *Journal of Parallel and Distributed Computing*, 73(12):1613–1626, 2013.

[74] Edmund Chow and Hartwig Anzt. *Asynchronous Iterative Algorithm for Computing Incomplete Factorizations on GPUs*, volume 9137 of *Lecture Notes in Computer Science*. Springer International Publishing, 2015.

[75] Nick Brown, J Mark Bull, and Iain Bethune. Solving large sparse linear systems using asynchronous multisplitting. *Partnership for advanced computing in Europe*, 2013.

[76] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha. Scalable work stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, 2009.

[77] Jacques M. Bahi, Sylvain Contassot-Vivier, and Raphael Couturier. Dynamic load balancing and efficient load estimators for asynchronous iterative algorithms. *IEEE Trans. Parallel Distrib. Syst.*, 16(4):289–299, 2005.

[78] Leonardo Dagum and Ramesh Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[79] Robert D Blumofe and Charles E Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.

[80] Harshitha Menon and Laxmikant Kalé. A distributed dynamic load balancer for iterative applications. In *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2013.

[81] EPCC. `https://www.epcc.ed.ac.uk`. Accessed: 01/01/2021.

[82] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, 1995.

[83] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[84] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[85] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[86] Sanghamitra Bandyopadhyay, Chris Giannella, Ujjwal Maulik, Hillol Kargupta, Kun Liu, and Souptik Datta. Clustering distributed data streams in peer-to-peer environments. *Information Sciences*, 176(14):1952–1985, 2006.